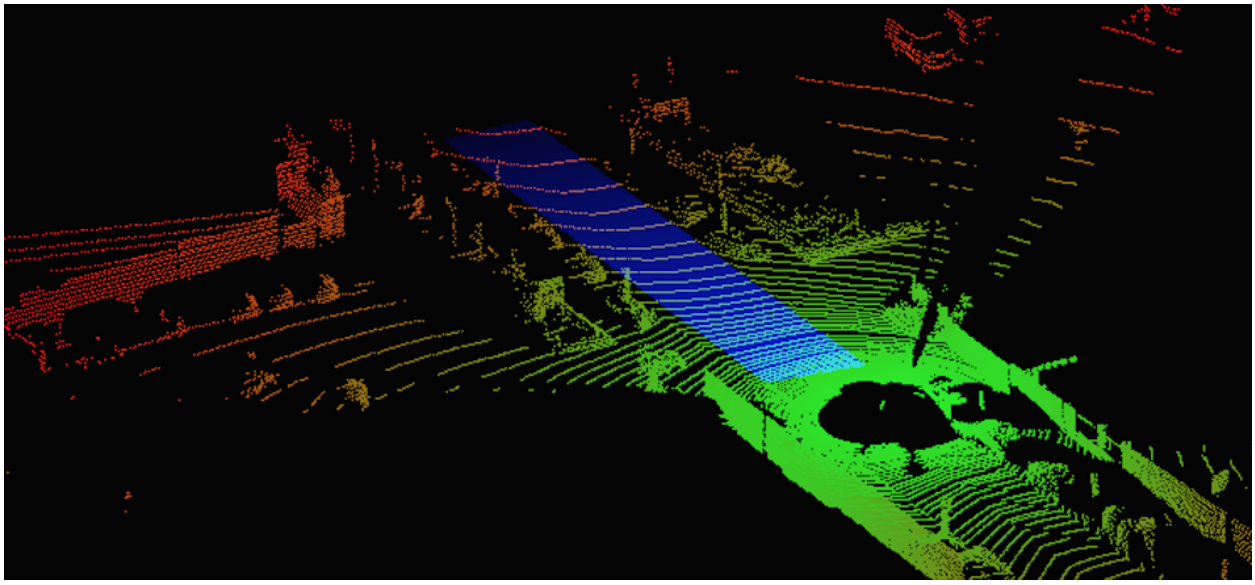


Unsupervised Feature Learning for Road Segmentation with Deep Neural Networks

Using Generative Models and Auxiliary tasks

Master's thesis in Complex Adaptive Systems

Benjamin Waubert & Simon Andersson

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Master's thesis EX039/2017

Unsupervised Feature Learning for Road Segmentation with Deep Neural Networks

Using Generative Models and Auxiliary tasks

BENJAMIN WAUBERT & SIMON ANDERSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Signals Processing and Biomedical Engineering
Signal Processing Group
Chalmers University of Technology
Gothenburg, Sweden 2017

Unsupervised Feature Learning for Road Segmentation with Deep Neural Networks Using Generative Models and Auxiliary tasks
BENJAMIN WAUBERT & SIMON ANDERSSON

Supervisor: Samuel Scheidegger, Zenuity
Examiner: Lennart Svensson, Electrical Engineering

Master's Thesis EX039/2017
Department of Electrical Engineering
Division of Signals Processing and Biomedical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Detections from a car mounted lidar sensor, color coded by distance from the sensor. The blue strip in front of the car is the area classified as road by the road segmentation method studied in this thesis.

Typeset in LaTeX
Gothenburg, Sweden 2017

Unsupervised Feature Learning for Road Segmentation with Deep Neural Networks Using Generative Models and Auxiliary tasks
BENJAMIN WAUBERT & SIMON ANDERSSON
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Deep learning techniques have lately shown promising results when applied to computer vision problems. An area benefiting from these techniques is the development of autonomous vehicles. Neural networks can be used to obtain an accurate perception of the vehicle surroundings, but need to be trained with large amounts of data with high quality annotations to achieve high performance. These annotations are generally expensive and time consuming to create. Hence there is a demand for reducing the amount of annotated data necessary for training an accurate model. One way to address this problem is by leveraging the structure present in the data itself, using unsupervised training methods. The main objective of this thesis is to explore such methods applied in the area of autonomous vehicles.

The primary focus of this thesis is the KITTI road detection task of performing semantic segmentation of road area. The thesis investigates different semi-supervised methods to perform this task by utilizing large unannotated datasets together with annotated datasets of limited size. Different types of generative models are explored, including generative adversarial networks and variational autoencoders. The generative models are used to perform feature extraction using unannotated data. The explored methods for unsupervised feature learning are quantitatively evaluated by comparison to different baseline methods. It is shown that unsupervised feature learning can be used to improve the performance of models for road segmentation.

Keywords: Autonomous vehicles, Deep Learning, Semi-supervised Learning, Generative Models, Semantic Segmentation.

Acknowledgements

We would like to thank our supervisor at Autoliv, Samuel Scheidegger, for his excellent counsel during the project. We would also like to thank our examiner, Lennart Svensson, for a lot of interesting discussions and helpful insights. In addition, we would like to thank Luca Caltagirone for attending our meetings and all his valuable input. Finally, we would like to thank Autoliv for the opportunity to do this thesis.

Contents

List of Figures

List of Tables

1

Introduction

In recent years the research of Advanced Driver Assisted Systems (ADAS) has intensified and the development of self-driving road vehicles has accelerated. Automotive companies and software companies are collaborating and competing to increase the level of autonomy in today's human operated vehicles. There are a variety of reasons why a higher level of automation is desirable. For instance, it is believed to increase traffic safety, because human errors can be reduced. In addition, increased autonomy could utilize infrastructure more effectively, which can reduce traffic congestion and bring down emissions.

An important component in the development of ADAS and autonomous vehicles is perception, i.e., the vehicle's ability to perceive its surroundings. Systems for perception need to solve tasks such as: localizing driveable area, tracking cars and pedestrians, and detecting road markings and road-signs. This is achieved through a combination of a sensor suite including cameras, radars and laser scanners; and advanced signal processing algorithms based on machine learning.

Recent developments of image processing by Artificial Neural Networks (ANN) have yielded models which perform object detection and classification on images with high accuracy [1]. One important reason for the success of image processing is the availability of large datasets with high quality annotations. For example, the ImageNet Large Scale Visual Recognition Challenge [2] (ILSVRC) dataset consists of over 1.3 million images with annotations on image-level and object-level into 1000 classes. Many successful deep learning applications utilize such datasets to learn good feature representations, which can then be utilized to improve performance on the original task, a process referred to as pre-training. Other data domains, such as lidar or radar data, lack large well annotated datasets and hence can't utilize this approach. This poses a challenge in the development of signal processing algorithms which are important for the advancement of ADAS and autonomous vehicles.

One promising alternative to pre-training is unsupervised feature learning and semi-supervised methods. These methods allow feature representations to be learned from unannotated data, which for many data domains is abundant. Recent advances in semi-supervised learning [3], [4] have achieved good performance on simple classification tasks, while utilizing only a few labeled training examples. At the same time,

new feature learning methods [5] allow high quality features to be trained without relying on annotations in non-image domains. An interesting avenue of research is how to scale these methods to more complex tasks, such as detection and segmentation, and if they could serve as a drop in replacement for Imagenet pre-training for other data types.

1.1 Purpose

This thesis investigates unsupervised methods for feature learning, i.e., methods for learning efficient feature representations from unlabeled data, and by extension improve the performance of some supervised task. To study the impact of unsupervised feature learning, a supervised task useful for automotive applications was chosen. In particular, the thesis focuses on the task of road segmentation, that is, segmenting the area in front of the vehicle into road and non-road. This task is an example of a semantic segmentation task, that is, each pixel is classified as one of a set of predefined classes, in this case, road or non-road. A visualization of road segmentation from image data can be seen in figure 1.1. Road segmentation is an important part of the vision pipeline for ADAS, in combination with object detection and localization methods it enables a clear perception of the vehicle surroundings. This in turn can support various tasks such as lane-keeping or driving path generation [6], and could in the future be the foundation for control systems in fully autonomous vehicles.

In this thesis, only methods which perform road segmentation from lidar sensor data have been studied. This choice is motivated mainly by the lack of large annotated lidar dataset to perform pre-training in this domain, making unsupervised methods a more interesting alternative. Furthermore, road segmentation using image data has been studied to a greater extent than using lidar data.

Unsupervised feature learning can be utilized to leverage unlabeled data and improve performance of supervised methods. Acquiring high quality annotated data is an expensive process, and the studied methods could reduce the amount of annotations necessary for training deep learning models. In particular, feature learning has the potential to improve model generalization when few annotations are available, similarly to Imagenet pre-training in the image domain. In this thesis, several promising feature learning methods are evaluated in the context of road segmentation from lidar data. The various methods are compared, both to each other and to a supervised baseline. This thesis attempts to quantify the potential gains of utilizing new unsupervised feature learning methods on the task of road segmentation from lidar data.

(a) Input image

(b) Road segmented image

Figure 1.1: A visualization of the road segmentation task on image data, each pixel in the input image is classified as road (blue) or non-road.

1.2 Contributions

The main contribution of this thesis is the implementation, evaluation and comparison of unsupervised feature learning methods on the task of road segmentation from lidar data. In particular, two methods based on generative modelling are compared, Generative Adversarial Networks (GAN) and Variational Autoencoders (VAE). These methods are used to learn efficient feature representations, by utilizing unannotated data, in models for road segmentation. This feature representation is then used in a supervised model. In this way, the performance gain from unsupervised feature learning is evaluated.

In addition, a process for learning feature representations by optimizing an auxiliary task is investigated. Specifically, the task of predicting the next frame in a sequence is evaluated as an unsupervised feature learning technique. All of the above approaches are evaluated on the KITTI road segmentation task, and are compared to an appropriate fully supervised baseline.

1.3 Related Work

The work in this thesis is based on, and enabled by, various sources of previous work. First and foremost, our work is enabled by advances in training deep neural networks.

Most notably, the normalization technique proposed in [7] significantly speeds up and stabilizes the training of deep neural networks. In addition, regularization techniques such as [8] simplify achieving good generalization in neural networks.

One of the methods for performing semantic segmentation of road area used in this thesis is based on the methods developed in [9]. This method was developed to perform semantic segmentation of road, using lidar data in a supervised setting. Results from [9] have been used as a baseline for comparison of results and their methods have been used for further development in a semi-supervised setting. The other method for performing road segmentation studied in this thesis is based on one of the most successful approaches for semantic segmentation, developed in [10]. These methods are modified to be applicable in a semi-supervised setting and be used with lidar data.

The unsupervised feature learning approaches studied in this thesis are based on recent advances in generative modelling. In particular, GAN [11] and VAE [12], as well as various modifications and extensions to these methods. More specifically, this thesis is based on work which applies these generative models in the semi-supervised setting [5]. This thesis evaluates the feasibility of applying these approaches to the task of road segmentation from lidar data.

1.4 Thesis Outline

The thesis is divided into six chapters: Introduction, Theory, Methods, Results for handwritten digits and Results for road segmentation, and finally Discussion. The second chapter covers the theory behind the concepts and frameworks used in this thesis; in this chapter all the theoretical background necessary for understanding the rest of the thesis is explained. The next chapter, Methods, describes how the theory is applied to our task, and how the studied methods are evaluated. In addition, background information about the problem and which data sources are used is presented. Next, the results of the used methods are presented in chapter 4 and 5. The results are shown for different variations of the problem and different experimental settings, where the results are compared using a set of evaluations metrics. Finally, a discussion of the results is presented.

2

Theory

This chapter provides an overview of the theory behind the methods utilized in this thesis. Necessary concepts are introduced and explained using theoretical derivations and simple examples. The concepts are connected to the purpose of the thesis and also motivate the work conducted. The chapter aims to give thorough theoretical background to understand all parts of the thesis work. First, a brief summary of statistical learning is given, establishing some terminology for later sections. Then, artificial neural networks and how to train them is covered, followed by a more in depth explanation of Convolutional Neural Networks (CNN) and the various building blocks which they consist of. Lastly, generative models are covered. In particular Variational Autoencoders (VAE) and Generative Adversarial Networks (GAN), with focus on their use in unsupervised feature learning.

2.1 Statistical Learning

In statistical learning the objective is to find a model that can perform a certain task. The model denoted $f(\mathbf{x})$ takes an input of explanatory variables and produces an output depending on the task. Using methods of statistical learning this model can be found. These methods utilize some observed data of explanatory variables $[\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}]$, and may or may not include corresponding response variables $[\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(n)}]$. This data is then used in a *learning method* to find an accurate model $f(\mathbf{x})$. One can say that the model is *trained* to solve a specific task using the observed data. The different learning methods can be divided into *supervised* and *unsupervised* learning methods, where the distinctions depend on what data is available and which task the method solves.

2.1.1 Supervised Learning

Supervised learning is a statistical learning method that is used when the data consists of both explanatory variables and corresponding response variables. The task is to find a model $f : \mathcal{X} \to \mathcal{Y}$ which maps an input of explanatory variables

$\mathbf{x}$ to an output prediction $\mathbf{y}$. The model is trained using data consisting of known pairs of explanatory and response variables $\{[\mathbf{x}_1, \mathbf{y}_1], [\mathbf{x}_2, \mathbf{y}_2], ..., [\mathbf{x}_N, \mathbf{y}_N]\}$ to infer the model parameters. The output is chosen to represent the task to be performed of the model. Example of tasks to be performed by a model can be

- **Classification**: Given the input data $\mathbf{x}$ the model should output an assignment to a specific category which the data belongs to. For example, given an input data $\mathbf{x}$ containing the height and weight of a person, a classification model could for example output the gender of the person. The output could be a binary variable representing 0 for male and 1 for female.
- **Regression**: Given the input data $\mathbf{x}$, the task of the model is to output a numerical value $\mathbf{y}$. For example, the input data can be the gender and height of a person and the model should then output a numerical value of the weight for the person.

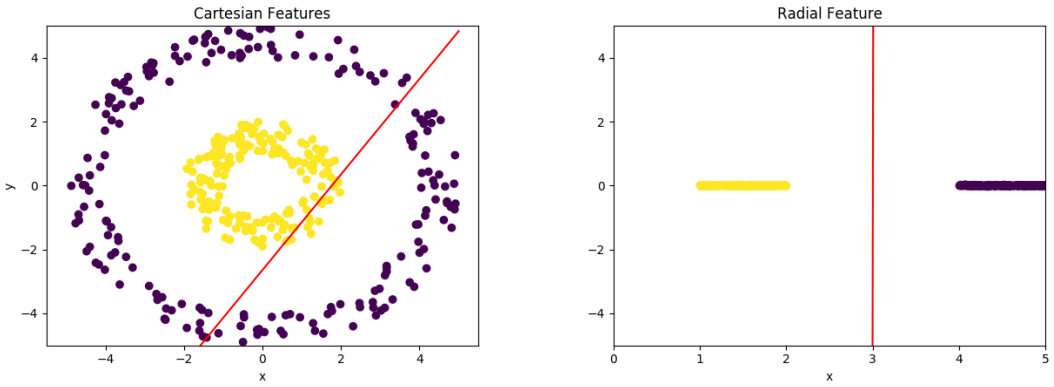
2.1.2 Unsupervised Learning

Unsupervised learning is a set of techniques that can be used when the observed data does not contain response variables $\mathbf{y}$. When the data only contains the explanatory variables $\mathbf{x}$, it is not possible to find a model that can predict an output of response variables. Instead unsupervised learning methods are used find structures and properties of the observed data. For example, *cluster analysis* is an unsupervised method, where the task is to find distinct clusters within the observed data [13].

2.1.3 Feature Learning

Feature learning is an unsupervised learning method where the explanatory variables are used to extract new, more useful features to represent the data. The idea is that the data can be represented by other feature variables that will be more descriptive of the data than the original explanatory variables. By using these feature variables the performance of statistical models could be improved, such as supervised models for classification or regression. The features can be learned by implementing a model that maps the raw input $\mathbf{x}$ to a new representation of latent feature variables $\mathbf{z}$. This model can be trained using only data of explanatory variables $\mathbf{x}$, without the need of prediction data $\mathbf{y}$.

A simple example of feature learning is shown in figure 2.1. Here the data consist of two classes distributed on concentric circles with different radius. If a linear classifier was to be trained to discriminate between the two classes it would have low performance with the original features of Cartesian coordinates (figure 2.1a). If the data is transformed to be represented by a feature that is the Euclidean distance to the origin, and a discriminator is trained on this feature it would perform significantly better. (figure 2.1b).



(a) Cartesian features **(b)** Radius feature

Figure 2.1: A linear discriminator applied to data of two classes distributed on two concentric circles. The figure to the left shows the discriminator marked by the red line applied to data using Cartesian features. The figure to the right shows the discriminator applied to data using radial features. The two figures illustrate that a linear discriminator performs better when applied to data with radial features.

2.1.4 Optimization Methods

For a learning method to be able to find an accurate model for a task to be solved, a quantitative measure of the performance is needed. The model can then be trained by adjusting its parameters to maximize this performance measure. This is usually done with some optimization method that solves

$$\underset{\theta}{\text{argmax}} \quad F(f_\theta(\mathbf{x}), \mathbf{y}). \tag{2.1}$$

Where $F(\cdot)$ is the performance measure and $\boldsymbol{\theta}$ the parameters of the model. A common approach to solve the optimization is to use the gradient descent method.

Gradient descent is an iterative method that sequentially updates the parameter values to optimize the objective function, by taking steps along the direction in which the gradient is largest. Gradient descent hence requires the performance metric F to be differentiable. The method is initialized at some starting parameter values $\boldsymbol{\theta}^{(t)}$. In every step, the parameters are updated by

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \gamma \nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}). \tag{2.2}$$

The parameter γ is called the learning rate and determines the size of the updating steps applied to $\boldsymbol{\theta}$. The sign in front of γ determines if the objective function is maximized (positive sign) or minimized (negative sign).

To get the exact gradient for the entire data set of n examples, the expectation over the empirical data distribution p_{data} needs to be computed

$$\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{data}}[\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y})] = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta}; \mathbf{x}^{(i)}, \mathbf{y}^{(i)}). \tag{2.3}$$

But computing this expectation is expensive, because it needs to be evaluated over all examples in the data set. Instead another method can be used, called *Stochastic Gradient Descent*. The gradient is then estimated by using a subset S_m that consists of m examples from the dataset. These examples are chosen by a random sampling from the data every time the gradient is computed. Using this estimation, the gradient is given by

$$\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) \approx \frac{1}{m} \sum_{\{i|\mathbf{x}^{(i)} \in S_m\}} \nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta}; \mathbf{x}^{(i)}, \mathbf{y}^{(i)}). \tag{2.4}$$

The standard error of this expectation estimation is $\frac{\sigma}{\sqrt{m}}$, meaning that the accuracy of the estimation will scale with the square root of the number of samples. The computational cost scales linearly with the number of samples, this motivates a smaller number of samples as a trade-off between estimation accuracy and computational cost. In the case of a very redundant data set it also makes sense to use a subset of the data, because similar examples in the data will give very similar contributions to the gradient [14].

2.2 Artificial Neural Networks

Artificial Neural networks, commonly referred to only as Neural networks, are a framework that can be used to solve different statistical learning tasks, such as classification or regression. Neural networks consist of multiple layers which can transform input data into new representations. The layers of a model consists of simple but non-linear operations, which when they are combined and stacked can form very complex functions. This property gives neural networks the ability to learn high level and abstract representations from data, that can be used for different tasks. By using optimization methods neural networks can be trained to accurately model complex relations.

2.2.1 Perceptrons

The simplest building block of a neural network model is the perceptron illustrated in figure 2.2. It is a simple computational unit which takes an input vector $\mathbf{x} \in \mathbb{R}^n$ and outputs a value y. A scalar multiplication is performed of the inputs to the perceptron and a vector of weights $\mathbf{w} \in \mathbb{R}^n$, and a bias term b is added. The value is then passed through an *activation function* $\phi(\mathbf{x})$ which gives the output

$$y = \phi(\mathbf{w}^T\mathbf{x} + b). \tag{2.5}$$

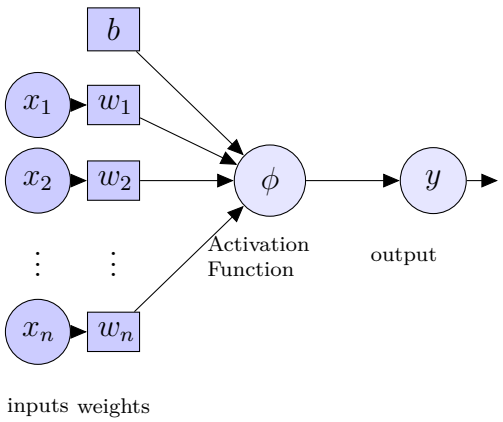


Figure 2.2: Illustration of a simple perceptron. The perceptron gets input $x_1, x_2 \dots x_n$ and computes the output y. The computation is done by using the weight parameters $w_1, w_2, \dots w_n$ and bias parameter b and the activation function ϕ by the equation $y = \phi(\mathbf{w}^T\mathbf{x} + b)$.

2.2.2 Fully Connected Network

By adding multiple perceptrons together a layer of a neural network is formed. Each perceptron then computes its output y from all the inputs from the previous layer

by equation (2.5). The size of the layer is determined by the number of perceptrons. Multiple layers can be stacked, where the output of each layer is used as input for the next. When every perceptron in a layer is connected to every perceptron in the next, the neural network is referred to as a *Fully Connected Network*. A visualization of such a network can be seen in figure 2.3. A fully connected network with L layers takes an input $\mathbf{x}$, gives an output $\mathbf{y}_{output}$ and is determined by the parameters $\boldsymbol{\theta} = \{[\mathbf{W}^{(1)}, \mathbf{b}^{(1)}] \ldots, [\mathbf{W}^{(L)}, \mathbf{b}^{(L)}]\}$. The matrices $\mathbf{W}^{(l)} \in \mathbb{R}^{(n_l)\mathrm{x}(n_{l-1})}$ represent the weights between the n_{l-1} perceptrons of layer $l-1$ and the n_l perceptrons of layer l. In particular, $w_{ij}^{(l)}$ is the weight connecting the output $(y_j^{(l-1)})$ of the j^{th} perceptron in layer $l-1$ to the i^{th} perceptron of layer l. The vectors $\mathbf{b}^{(l)} \in \mathbb{R}^L$ represent the bias values for the perceptrons in layer l. With this notation the output of layer l in the network is given by:

$$\mathbf{y}^{(l)} = \boldsymbol{\phi}(\mathbf{W}^{(l)}\mathbf{y}^{(l-1)} + \boldsymbol{b}^{(l)}), \quad \mathbf{W}^{(l)} \in \mathbb{R}^{(n_l)\mathrm{x}(n_{l-1})}. \tag{2.6}$$

The final output y of the fully connected network is then given by the function $f_{\boldsymbol{\theta}}(\mathbf{x})$ that consists of nested output operations for all layers, starting from the input layer:

$$\mathbf{y}_{output} = f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\phi}(\mathbf{W}^{(L)}\boldsymbol{\phi}(\mathbf{W}^{(L-1)}\boldsymbol{\phi}(\ldots) + \boldsymbol{b}^{(L-1)}) + \boldsymbol{b}^{(L)}). \tag{2.7}$$

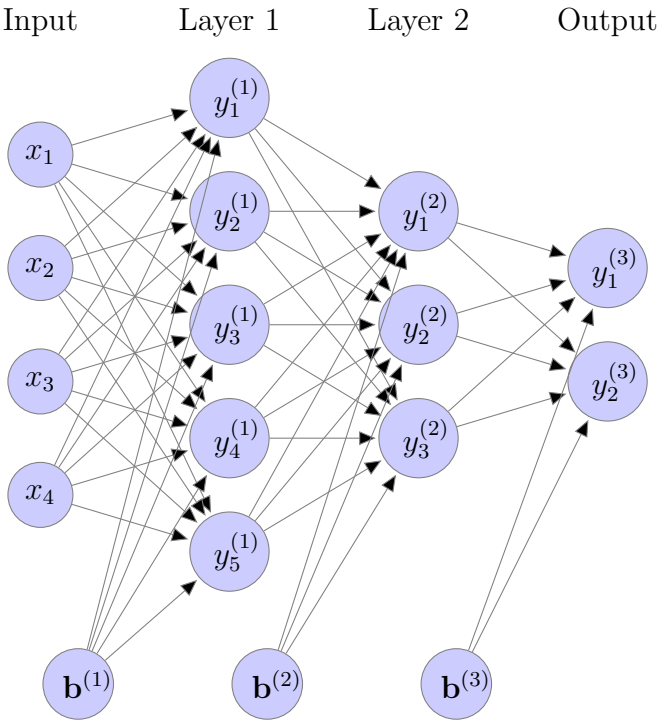


Figure 2.3: Visualization of a neural network with 2 hidden layers. $y_j^{(i)}$ represents the output of perceptron j in layer i.

The bias parameter $\mathbf{b}$ can be replaced by a weight parameter. By adding an additional input neuron with constant value of 1, the bias vector can be replaced by weights connecting this neuron to all output neurons. This will make the notations of the equations easier, and the bias parameters $\mathbf{b}$ will be contained in the weight matrix $\mathbf{W}$.

2.2.3 Activation Functions

The activation functions $\phi(\cdot)$ used in neural networks are generally non-linear functions and can take many different forms. The addition of activation functions is necessary for enabling a neural network to model non-linear relations between variables. Without them, neural networks can only represent linear transformations. By introducing non-linearities between neural network layers, much more complex relations can be approximated.

There are many different functions that can be used to introduce non-linearities in neural networks, some of the most commonly used are:

$$\text{rectified linear unit(ReLU):} \quad \phi_{ReLU}(x) = \max(0, x). \tag{2.8}$$

$$\text{leaky rectified linear unit:} \quad \phi_{lReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}. \tag{2.9}$$

$$\text{exponential linear unit(ELU):} \quad \phi_{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}. \tag{2.10}$$

$$\text{tanh function:} \quad \phi_{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.11}$$

$$\text{sigmoid function:} \quad \phi_{sigmoid}(x) = \frac{1}{1 + e^{-x}}. \tag{2.12}$$

$$\tag{2.13}$$

A visualization of these functions can be seen in figure 2.4.

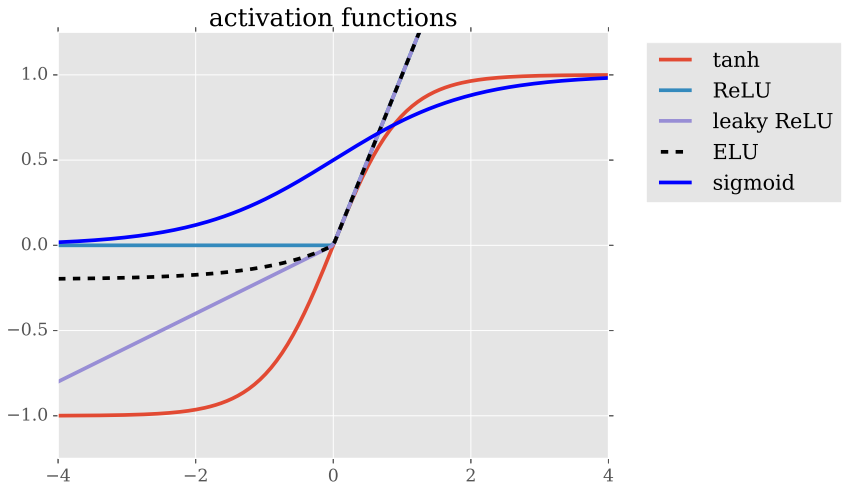


Figure 2.4: Visualization of common activation functions.

2.2.4 Network output

Depending on the task that a network is assigned to solve, the output layer has different designs. By adapting the shape of the output by the number of neurons and which activation function is being used, the network can be designed for different tasks. If a regression problem is being solved, the activation of the output neurons is a linear function, enabling the output to take any real value. The number of output neurons is determined by the number of response variables of the problem that is being solved.

If the task is a classification problem, the output should be one or multiple assignments to different classes. In this case, the activation function of the output neurons will be the softmax function. This function will give probability estimations for the assignment to each class and is given by

$$\hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_k], \quad \hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}. \tag{2.14}$$

Where z_i is the activation of i:th neuron given by the outputs of the previous layer and the weights connected to the last layer. The number of output neurons of the last layer denoted by k is given by the number of classes in the task to be solved. The output $\hat{\mathbf{y}}$ is a vector of length k which sums to unity and y_i gives the estimation of the probability of the sample belonging to class i.

2.2.5 Loss minimization

Neural networks are generally trained by using stochastic gradient descent, see section 2.1.4, to minimize a loss function. In the supervised setting, the model is optimized to minimize a loss function which is designed to model the cost of mistakes made. A loss function C is defined as a function of the network output. The most common choice when performing classification is the cross entropy loss:

$$C(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i \in \mathcal{C}} [y_i \log(\hat{y}_i)]. \tag{2.15}$$

where $\mathcal{C}$ is the set of classes, $\hat{y}_i$ denotes the network's output of softmax probabilities by equation 2.14 for class i and y_i denotes the class label. Both the network output and target are here assumed to be vectors which length correspond to the number of classes. In addition, it is assumed that the label element corresponding to the correct class equals one, and the remainder zero. Clearly, this expression will take its smallest value when $\boldsymbol{y'} = \boldsymbol{y}$. For the case of regression, the mean squared error is often used:

$$C(\boldsymbol{y'}, \boldsymbol{y}) = \frac{1}{m} \sum_{i=1} ||\boldsymbol{y}'_i - \boldsymbol{y}_i||_2^2, \tag{2.16}$$

The procedure for training a neural network consists of three steps, which are repeated for a number of iterations. First, data is fed forward through the network and

the loss is calculated. The gradient of the loss function is then calculated with respect to the neural network parameters using the method of backpropagation, which is covered in the following section. Once the gradient is calculated it can be used by a gradient descent algorithm to update the model parameters in the direction which minimizes the loss. Generally, data is fed in small batches, and parameters are updated incrementally for each batch.

2.2.6 Backpropagation

Backpropagation is a method for calculating the gradient of the loss function with respect to the parameters of a neural network. The gradient are calculated layer by layer starting from the last layer (L). Using a recursive process and the properties of the chain rule, the gradient of the loss function can be propagated backwards in the network layer by layer. Starting from the output layer the loss function is given as a function of the layer parameters by

$$C(\boldsymbol{y}^L) = C(\phi(\mathbf{W}^L\mathbf{a}^{L-1})) = C(\phi(\mathbf{z}^L)). \tag{2.17}$$

The gradient with respect to an activation z_j^L is computed by

$$\frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\phi'(z_j^L) = \delta_j^L. \tag{2.18}$$

The gradient with respect to the layer parameters and a weight w_{ij} is

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{ij}^L} = \frac{\partial C}{\partial a_j^L}\phi'(z_j^L)a_i^{(L-1)} = \delta_j^L a_i^{(L-1)}. \tag{2.19}$$

For the preceding layer the loss function with respect to the weights is

$$C(\mathbf{W}^{(L-1)}) = C(\phi(\mathbf{W}^{(L)}\phi(\mathbf{W}^{(L-1)}\mathbf{a}^{(L-2)})) = C(\phi(\mathbf{W}^{(L)}\phi(\mathbf{z}^{(L-1)})). \tag{2.20}$$

Using the chain rule, the gradient with respect to a activation $z_i^{(L-1)}$ is given by

$$\frac{\partial C}{\partial z_i^{(L-1)}} = \sum_{j=1}^{n_L}\frac{\partial C}{\partial z_j^L}\frac{\partial z_j^L}{\partial z_i^{(L-1)}} = \sum_{j=1}^{n_L}\delta_j^L w_{ij}\phi'(z_i^{(L-1)}) = \delta_i^{(L-1)}. \tag{2.21}$$

For a weight parameter w_{ki} connecting neuron k in layer $L-2$ to neuron i in layer $L-1$ the gradient can be computed by

$$\frac{\partial C}{\partial w_{ki}^{(L-1)}} = \frac{\partial C}{\partial z_i^{(L-1)}}\frac{\partial z_i^{(L-1)}}{\partial w_{ki}^{(L-1)}} = \delta_i^{(L-1)}a_k^{(L-2)}. \tag{2.22}$$

These computations can recursively be propagated backwards in the network to compute all the gradients for all parameters in every layer.

The gradients are used to update the parameters of the network using a gradient descent method by the equation

$$w_{ij}^{l} \rightarrow w_{ij}^{l} + \gamma \frac{\partial C}{\partial w_{ij}^{l}}, \tag{2.23}$$

where γ is the learning rate and w_{ij}^{l} the weight connecting neuron i of layer $l-1$ to neuron j of layer l.

2.2.7 Parameter Initialization

When a network is trained using backpropagation the parameters of the network need to be initialized before the training procedure. There are several methods with which the initialization can be done. The optimization problem of training neural networks is generally non-convex, and therefore multiple local optima can exist. The starting point is then important to make the algorithm converge to a good local optimum.

A simple choice is to initialize the parameters by a random process. All the initial parameter values are sampled from some distribution. This distribution is usually chosen to be a Gaussian distribution with zero mean that is truncated at some value, to avoid too large parameter values. But more carefully chosen initialization can be beneficial for the training of the network. Xavier initialization has been shown to be an effective method for this [15]. With this initialization the variance of the activations are maintained through the layers, preventing exploding and vanishing gradients.

For parameters initialized using Xavier initialization with an uniform distribution, the initial weights are distributed as

$$w_{ij}^{(l)} \sim \mathcal{U}\left(-\sqrt{\frac{6}{(n_{l-1}+n_l)}}, \sqrt{\frac{6}{(n_{l-1}+n_l)}}\right). \tag{2.24}$$

Where n_{l-1} and n_l are the number of neurons in the layers. When a Gaussian distribution is used the weights are initialized by the distribution

$$w_{ij}^{(l)} \sim \mathcal{N}\left(0, \sqrt{\frac{3}{(n_{l-1}+n_l)}}\right), \tag{2.25}$$

and the bias parameters will be initialized to zero.

2.2.8 Weight Decay

Weight decay or regularization is a method for constraining the effective number of free parameters in a network to avoid overfitting and improve generalization. The

method consists of penalizing large weights by adding the L2-norm of the weights to the loss function as:

$$\tilde{C}(x) = C(x) + \frac{\lambda}{2}\sum_{l=1}^{L}||\boldsymbol{W}^{(l)}||_2^2, \tag{2.26}$$

where λ is a parameter controlling how strong the regularization is. This method biases the model towards simpler functions and therefore often results in classification rules which generalize better. The method is known as weight decay because the derivative of the gradient regularization term is:

$$\frac{\partial}{\partial w_{i,j}^{(l)}}\frac{\lambda}{2}\sum_{l=1}^{L}||\boldsymbol{W}^{(l)}||_2^2 = 2\lambda w_{i,j}^{(l)}, \tag{2.27}$$

and hence every weight parameter decays by a term proportional to its size for every step of gradient descent.

2.2.9 Dropout

Dropout [8] is a method to prevent neural networks from overfitting and decrease generalization error. Dropout is performed by randomly dropping a subset of neurons, and all their connections, for each step of gradient descent. This can be seen as sampling a thinned network from the neural network one wishes to train, and feeding data through that instead of the entire network. In practice the network is trained with a dropout probability $1-p$ on a number of neurons, i.e., each such neuron is zeroed with a probability $1-p$. The weights connecting from these neurons will get a gradient of zero and will not be updated in the training step. When the network is not in training and instead used for prediction, the activation values of the neurons will be scaled by the factor p. Compensating for that only a fraction p of neurons were available during the training. This process can roughly be thought of as creating an 'artificial' ensemble of neural networks and averaging their output, creating a more robust model and counteracting overfitting.

2.2.10 Batch Normalization

Batch normalization [7] is a method for normalizing intermediate activations of deep neural networks, leveraging the fact that data is generally fed in small batches. The normalization step is implemented as a part of the network and has been proved to significantly increase convergence rate, allow for higher learning rates and decrease sensitivity to initialization, in particular for deep architectures.

The normalization is performed for all the activations in the layers of the network. For every mini-batch that is fed through the network during training the mean and

variance are computed for all the activations by:

$$\mu_{i,B}^{(l)} = \mathbb{E}_{x\sim B}\left[z_i^{(l)}\right] \qquad \text{and} \qquad \sigma_{i,B}^{l} = \sqrt{\mathbb{E}_{x\sim B}\left[\left(z_i^{(l)} - \mu_{i,B}^{(l)}\right)^2\right]}. \tag{2.28}$$

where $\mu_{i,B}^{l}$ and $\sigma_{i,B}^{l}$ denotes the mean and variance for the i^{th} activation in the l^{th} layer for batch B. These values are then used to normalize the output as:

$$\hat{\boldsymbol{z}}^{(l)} = \frac{\boldsymbol{z}^{(l)} - \boldsymbol{\mu}_B^{(l)}}{\sqrt{(\boldsymbol{\sigma}_B^{(l)})^2 + \epsilon}}, \tag{2.29}$$

Where ϵ is a very small number added to avoid a vanishing denominator. To ensure that the normalization transform does not limit the expressive capacity of the network, the output is then rescaled and shifted using two learned parameters γ and β as:

$$\tilde{\boldsymbol{z}}^{(l)} = \gamma\hat{\boldsymbol{z}}^{(l)} + \beta. \tag{2.30}$$

These parameters are treated as model parameters and are trained through stochastic gradient descent and backpropagation. At inference time the mini batch mean and variance are replaced by population statistics, that is the mean and variance of the intermediate outputs for the entire dataset.

2.2.11 Layer Normalization

Layer Normalization is an in-layer normalization method very similar to batch normalization. The major difference is that layer normalization normalizes each input sample independently and is instead reduced over all the neurons in a layer. This means normalization coefficients are not shared between samples in a batch, but instead by all neurons in a layer. For a layer with N neurons the normalization coefficients are calculated as:

$$\mu^{(l)} = \frac{1}{N}\sum_{i=1}^{N} z_i^{(l)} \qquad \sigma^{(l)} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(z_i^{(l)} - \mu^{(l)})^2} \tag{2.31}$$

The data is then normalized as in equation 2.29. One advantage of this method of normalization is that it does not introduce any interaction between the samples within a batch.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNN) are a class of neural networks with properties of using spatially structured neurons, local connectivity and weight sharing. A CNN

consists of several convolutional layers, pooling layers (see sec 2.3.4) and in some cases fully connected layers.

In contrast to fully connected layers, convolutional layers are arranged in multidimensional volumes of neurons. In general, the layers will consists of 3-dimensional volumes with a height, width and depth. Each neuron is connected to a small region of the previous layer or input, but always covers the entire depth. In addition, all neurons in the same depth layer share weights.

Convolutional Neural networks work particularly well on inputs with spatially correlated structure, such as images. When working with high dimensional data, in particular images, the number of weights required to build fully connected layers become prohibitive. Convolutional layers enable more parameter-efficient architectures by using weight sharing. The weight sharing also introduces translation invariance, which is desirable in many image classification tasks.

2.3.1 Convolutional Layer

The most basic building block of a CNN is the convolutional layer. In a convolutional layer, each node is connected only locally to its input layer. Instead of connecting every input neuron to every output neuron with unique weights, a set of *kernels* or *filters* are convolved with the input to create the next layer.

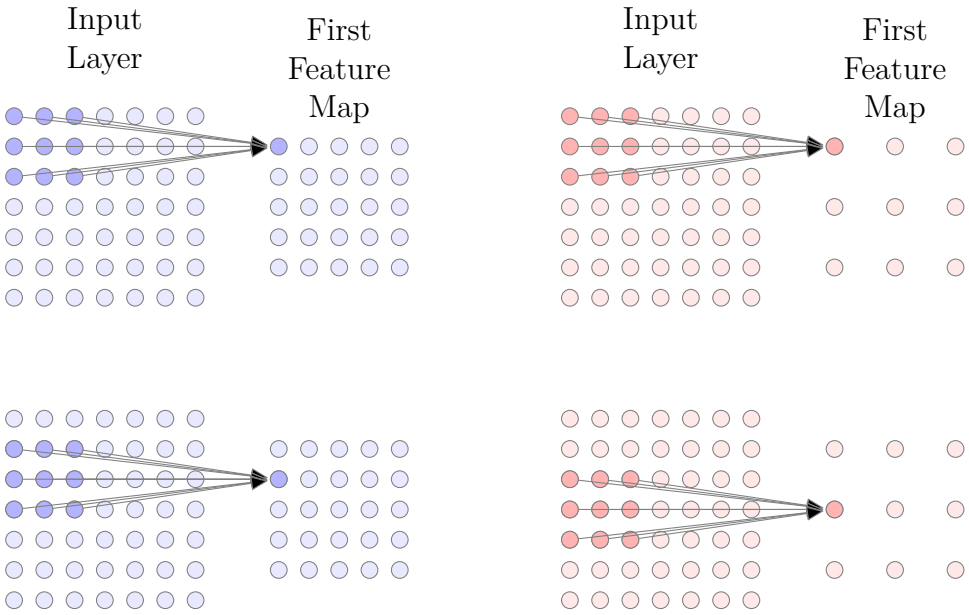
More formally, given M kernels of size $K_x \times K_y \times D$ and a layer input with D channels, the weight matrix $\boldsymbol{W}$ is of dimension $K_x \times K_y \times D \times M$. The output of a neuron in layer l is then defined as:

$$a^{(l)}_{k,n,m} = \boldsymbol{\phi}(z_{k,n,m}) = \boldsymbol{\phi}(b + \sum_{i=0}^{K_x}\sum_{j=0}^{K_y}\sum_{d=0}^{D} w^{(l)}_{i,j,d,m} a^{(l-1)}_{k+i,n+j,d}). \tag{2.32}$$

An example of how a $3 \times 3 \times 1$ filter is convolved with a $7 \times 7 \times 1$ input volume can be seen in figure 2.5a. The important thing to note is that the same 3x3x1 weights are used to create every unit in the first layer of the convolutional layer's output volume, meaning only 9 parameters are used in this layer. Unit depth layers of neurons of this type are commonly referred to as *feature maps*. Multiple feature maps are created by convolving multiple filters with the input volume and then stacked along the third dimension, creating a so called *feature volume*.

2.3.2 Stride

In figure 2.5a the filter is moved over the input one unit at a time, this is referred to as convolution with a *stride* of 1. Other stride lengths can be used, for instance



(a) Stride 1 **(b)** Stride 2

Figure 2.5: Visualization of a the local connectivity of a convolutional neural network with a input size of $7 \times 7 \times 1$ and a kernel size of $3 \times 3 \times 1$, using two different strides.

in figure 2.5b the filter is moved over the input with a stride length of 2. There are various reasons for using a larger stride length. Most commonly, it is used as a technique for down-sampling, as it helps reduce the size of feature volumes and the parameters necessary in later layers.

2.3.3 Padding

Padding is a property that can be included in a convolutional operation for different purposes. Padding simply means adding extra values around the input feature map before applying a convolution. The values can be either zeros or a reflection of the values in the input map. When applying a convolution without padding the edges of the input feature map are only convolved with the edges of each filter, hence the operations output will be slightly smaller than the input. A common procedure for padding is *half padding*, where each side of the input is padded with a number of zeros corresponding to half the kernel size, rounded down to the nearest integer

$$p = \lfloor \frac{K}{2} \rfloor. \tag{2.33}$$

Where p denotes by how many units the input is padded, and K denotes the kernel size. Half padding maintains the feature map size through the convolution operation, that is, the output will have the same size as the input. Half padding is used for all convolutional layers in this thesis.

2.3.4 Pooling

In addition to convolutional layers CNNs often contain pooling layers, these are layers which down-sample a feature volume to a more compact representation. Each neuron in a pooling layer considers a region of $K_p \times K_p$ activations in the previous layer and summarizes them. Generally the input is divided into a number of disjoint $K_p \times K_p$ regions, which each are pooled into one output. The most common type of pooling is max-pooling , where each neuron in the pooling layer returns the max value in its input region

$$a^{(l)}_{k,n,m} = \max_{i=[0,..K_p-1],j=[0,...,K_p-1],} a^{(l-1)}_{k+i,n+j,m}. \tag{2.34}$$

Average pooling is another operation which instead computes the average value in pooling region by

$$a^{(l)}_{k,n,m} = \frac{1}{K_p^2} \sum_{i=0}^{K_p-1} \sum_{j=0}^{K_p-1} a^{(l-1)}_{k+i,n+j,m}. \tag{2.35}$$

2.3.5 Unpooling

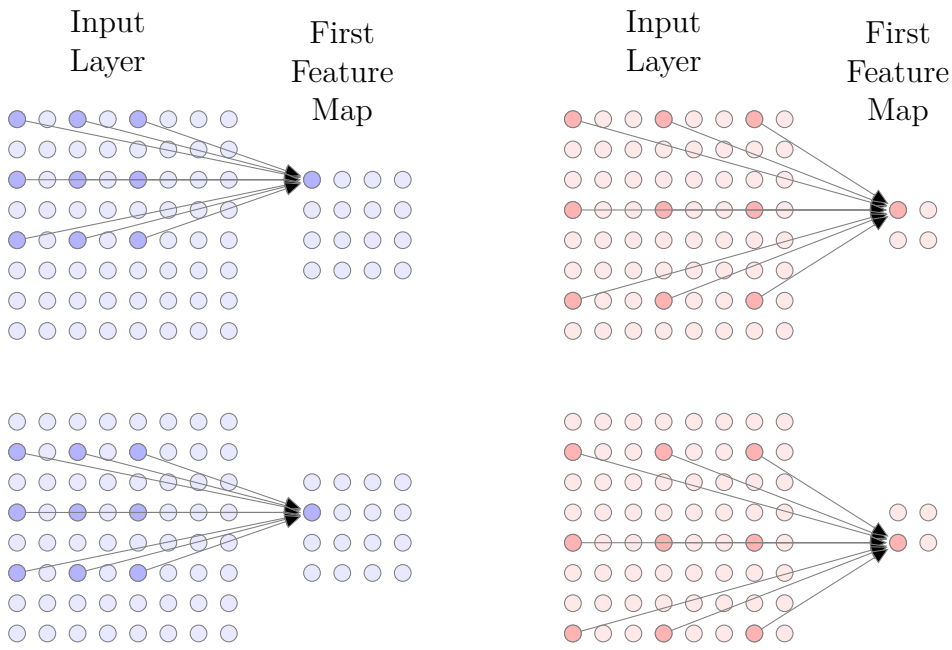
Unpooling operations can be used in a neural network to perform upsampling. The operations maps a region of size $K_p \times K_p$ from the input map to a larger region in the output map $K_o \times K_o$, $K_o > K_p$. For example an average-unpooling operation with stride 2, maps each value from the input map to 2×2 disjoint regions in the output map. Where the input values are averaged over the 2×2 positions of the output. This results in a 2 times upsampling of the input. A max-unpooling operation can also be used, to reverse a max pooling operation. Each value is then assigned a position in the output region corresponding to the max position from the pooling operations, the remaining positions in the region are set to a constant.

2.3.6 Dilated Convolution

Dilated convolutions [16] is a technique for applying the filters in a CNN in a dilated fashion. An illustration of dilated convolution can be seen in figure 2.6. This method allows for networks to have a larger receptive fields(each neuron is connected to an larger area of the input) without increasing the number of parameters. A unit in a dilated convolutional layer is defined as:

$$a^{(l)}_{k,n,m} = \phi(z^{(l)}_{k,n,m}) = \phi(b + \sum_{i=0}^{K_x} \sum_{j=0}^{K_y} \sum_{d=0}^{D} w^{(l)}_{i,j,d,m} a^{(l-1)}_{k+\tau i,n+\tau j,d}). \tag{2.36}$$

Where τ is the dilation factor, which controls by how many units the filters are diluted when applied to the input volume.



(a) Dilation factor 2 **(b)** Dilation factor 3

Figure 2.6: Visualization of dilated convolutional layer with a input size of $7 \times 7 \times 1$ and a kernel size of $3 \times 3 \times 1$, using two different dilation factors.

2.3.7 Transposed Convolution

Another operation that can be used in a CNN is the transposed convolution, also called deconvolution. This operation can be thought of as the opposite of the convolutional operator. Analogously to how a strided convolution can be used as a trainable down-sampling operation, strided transposed convolution can be used as a trainable up-sampling operation.

In a regular convolution a region of the input is mapped to a single value of the output by a weight matrix, visualized in figure 2.5a. In a transposed convolution, a single value of the input is mapped to a region of the output by a weight matrix. The input value is element wise multiplied with the matrix and the resulting matrix is added to a region in the output map. Regions in the output from different input values can overlap and their values are then added in the overlapping elements. In figure 2.7 an example of the transposed convolution operation is shown.

The transposed convolution can also be described using a expanded matrix of the weights. In a simple case with a 3×3 size input and a 2×2 kernel the regular convolution can be describe by a regular matrix operation

$$\underset{4\times1}{\mathbf{y}} = \underset{4\times9}{\mathbf{C}} \quad \underset{9\times1}{\mathbf{x}}, \tag{2.37}$$

where the matrix $\mathbf{C}$ consists of the weights from the kernel

$$\mathbf{C} = \begin{pmatrix} w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 & 0 \\ 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 \\ 0 & 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} \end{pmatrix}. \tag{2.38}$$

The output map is given by reshaping the vector $\mathbf{y}$. For the transposed convolution with the same weight matrix the corresponding matrix operation is

$$\underset{9\times1}{\tilde{\mathbf{x}}} = \underset{9\times4}{\mathbf{C}^T} \quad \underset{4\times1}{\mathbf{y}}. \tag{2.39}$$

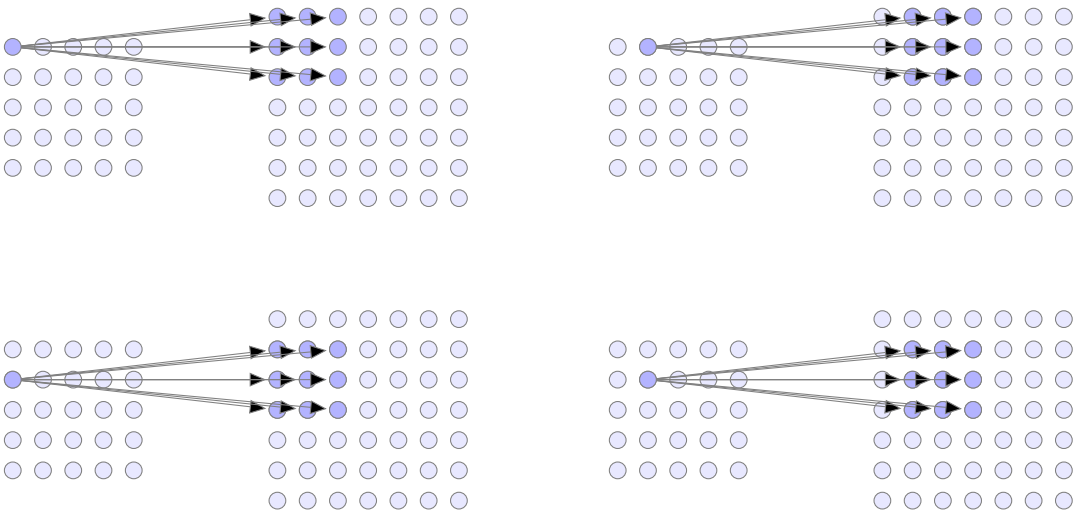


Figure 2.7: Visualization of a transposed convolution with kernel size 3×3, stride 1 and no padding. Each input value is element wise multiplied with the weight matrix and added to the corresponding region of the output.

2.3.8 Transfer Learning

The simplest way of initializing a neural network is to sample the weights from some distribution, often from a normal distribution. Transfer learning is a technique for training neural networks in a more data-efficient manner, by transferring a set of weights from another network and using those as a starting point for the training. Weights can be transferred either from a network trained supervised on another dataset, referred to as pre-training, or from a network trained by unsupervised feature learning. Transfer learning can speed up learning significantly because the entire network does not need to be retrained, and can also improve generalization.

The intuition behind why this works is that the filters learned in early layers of a CNN detect generic features such as edges and corners, and that these filters hence are applicable to other tasks. In transfer learning these features are learned from data similar to data used in the intended task to be solved, where there might be

transferable information between the tasks. The methods can also be used on data not very similar to the original task, which is a more general formulation called self-taught learning [17]. The advantages of applying these methods is that more data can be used, which can be beneficial for the learning task.

The procedure for training neural networks with transfer learning is as follows. First, layers 1 to l are initialized with pre-trained weights, the remaining layers are initialized randomly. The randomly initialized layers are then trained to convergence with the previous layers fixed, this is done to make sure the learned feature representation is conserved and avoid propagating noisy gradients back through it. Then the entire network is trained, in a step called fine-tuning. The number of layers which can be reused depend on the similarity of data used for pre-training and training, and how much training data is available.

2.4 Generative Models

Generative models are a class of probabilistic models which aim to learn a joint distribution over a set of variables, either as an explicit density or by enabling sampling from this distribution. The parameters of a generative model are generally inferred using a set of observations of the modeled variables. Generative models can be applied in a variety of ways, the focus of this thesis will be on their use as models for learning efficient representations from unannotated data. The goal is then to leverage these representations to improve performance on supervised tasks.

2.4.1 Density Estimation

The majority of generative models perform some type of density estimation, that is, the model aims to approximate the distribution of data p_{data}. This can be done either explicitly as a density $p_{model} \approx p(x)$, or implicitly through creating a model which can generate samples from an approximation of the data distribution. The majority of generative models are based on Maximum Likelihood Estimation (MLE), and the methods for generative modelling covered in this thesis can all be described as MLE based methods [18].

MLE is a method applied to a model that estimates a probability distribution. The model is parameterized by $\boldsymbol{\theta}$ and the likelihood is the probability that the model assigns the training data

$$\text{likelihood} = \prod_{i=1}^{m} p_{model}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{2.40}$$

The MLE approach is then to maximize this likelihood with respect to the model parameters

$$\begin{aligned}\boldsymbol{\theta}^* &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{model}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log \left(p_{model}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \right).\end{aligned} \tag{2.41}$$

Performing the MLE is equivalent to minimizing the Kullback-Leibler divergence [18].

Kullback-Leibler divergence is a measure of how much a distribution diverges from another. Given two distributions $p_{data}(\boldsymbol{x})$ and $p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$, the Kullback-Leibler divergence from $p_{data}(\boldsymbol{x})$ to $p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$ is denoted $D_{KL}(p_{data}(\boldsymbol{x})||p_{model}(\boldsymbol{x}; \boldsymbol{\theta}))$ and computed by

$$\begin{aligned}D_{KL}(p_{data}(\boldsymbol{x})||p_{model}(\boldsymbol{x}; \boldsymbol{\theta}))) = \mathbb{E}_{x \sim p_{data}(\boldsymbol{x})} \left[\log \frac{p_{data}(\boldsymbol{x})}{p_{model}(\boldsymbol{x}; \boldsymbol{\theta}))} \right] = \\ \int_{-\infty}^{\infty} p_{data}(\boldsymbol{x}) \log \frac{p_{data}(\boldsymbol{x})}{p_{model}(\boldsymbol{x}; \boldsymbol{\theta}))} dx.\end{aligned} \tag{2.42}$$

The divergence is a measure of the difference between the two distributions and is always non-negative. If the two distributions are equal the divergence is zero. The measure is also asymmetric and $D_{KL}(Q||p_{model}(\boldsymbol{x}; \boldsymbol{\theta}))) \neq D_{KL}(p_{model}(\boldsymbol{x}; \boldsymbol{\theta}))||Q)$. The Kullback-Leibler divergence can be used to evaluate an approximate model of a probability distribution against the true distribution. Using the model $p_{model}(\boldsymbol{x}; \boldsymbol{\theta}))$ and the true distribution $p_{data}(\boldsymbol{x})$.

In practice, the true distribution $p_{data}(\boldsymbol{x})$ is often not available, instead a dataset consisting of N samples is used. From this dataset an empirical distribution $\hat{p}_{data}(\boldsymbol{x})$ is defined, which places weight on exactly the points present in the dataset.

2.5 Variational Autoencoders

A method that can be used to create generative models is Variational Autoencoders (VAE). It is a model for approximating parameters in a probabilistic model by parameter inference. The method is based on the theory of variational Bayesian methods, which are techniques to approximate intractable integrals in Bayesian inference.

Assuming there is some observed data that consists of N i.i.d samples $\{\mathbf{x}^{(\mathbf{i})}\}_{i=1}^{N}$. This data is assumed to have been generated from some random process, where the

problem is to find this random process and the parameters of the model. This can be done with an Bayesian approach where the model of the random process is assumed to consist of a latent variable $\mathbf{z}$ with a prior distribution $\mathbf{z} \sim p(\mathbf{z})$ and the data is generated from the conditional distribution $\mathbf{x} \sim p_\theta(\mathbf{x}|\mathbf{z})$. The parameters $\boldsymbol{\theta}$ of the conditional distribution are the ones to be inferred by the data and the inference model. The parameters are implemented in the model by a deterministic function $f(\mathbf{z}; \theta)$, mapping the random latent variable $\mathbf{z}$ to the distribution parameters for $\mathbf{x} \sim p(\mathbf{x}|f(\mathbf{z}; \theta))$. Usually the prior distribution for $\mathbf{z}$ is assumed to be a Gaussian $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ and the distribution of $\mathbf{x}$ is assumed to be a Gaussian distribution with fixed variance. The latent variable $\mathbf{z}$ is mapped to the mean of the distribution for $\mathbf{x}$ and the probabilistic model for the data is

$$p_\theta(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}; f(\mathbf{z}; \theta), \sigma^2\mathbf{I}). \tag{2.43}$$

The parameter inference is usually done by MLE, see section 2.4.1. With the latent variable $\mathbf{z}$ introduced to the model, the likelihood needs to be computed for the marginal distribution of $\mathbf{x}$, and the resulting maximum likelihood equation becomes

$$\arg\max_{\boldsymbol{\theta}} p_\theta(\mathbf{x}) = \arg\max_{\boldsymbol{\theta}} \int p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}. \tag{2.44}$$

With a fairly complicated mapping function $f(\mathbf{z}; \boldsymbol{\theta})$, the integral becomes intractable to compute analytically and the maximum can not be found. An approach to address this problem is to approximate the distribution and then use optimization methods to find the maximum of the objective function and the parameters $\boldsymbol{\theta}$ of the model. This estimation can be done by sampling from $z_i \sim p(z)$ and approximate the likelihood function by $p_\theta(\mathbf{x}) \approx \frac{1}{n}\sum_{i=1}^{n} p_\theta(\mathbf{x}|\mathbf{z}_i)p(\mathbf{z}_i)$. But with high dimensional data the number of samples needs to be extremely large to get a good approximation. VAEs can be used to make the sampling procedure more effective and get a better approximation of the distribution.

Instead of sampling from the prior distribution $p(\mathbf{z})$, a better choice would be to sample from the posterior distribution $p(\mathbf{z}|\mathbf{x})$. The samples $\mathbf{z}_i$ will then have higher probability of generating samples $\mathbf{x}_i$ that matches the data and fewer samples are needed to make a good approximation of the likelihood function $p_\theta(\mathbf{x})$. To find a posterior distribution Bayes rule can be used

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{z}|\mathbf{x})p(\mathbf{z})}{p(\mathbf{x})}. \tag{2.45}$$

But this is again intractable due to the marginal probability of $\mathbf{x}$. Instead a recognition model $q_\phi(\mathbf{z}|\mathbf{x})$ is introduced to approximate this posterior distribution.

The recognition model is a mapping from the data space $\mathbf{x} \in \mathcal{X}$ to a distribution of $\mathbf{z}$. The distribution is usually chosen to be a multivariate Gaussian distribution with diagonal covariance matrix. The recognition model maps the input data to a vector of means $\boldsymbol{\mu}$ and a vector of standard deviations $\boldsymbol{\sigma}$

$$p(\mathbf{z}|\mathbf{x}) \approx q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_\phi(\mathbf{x}), \boldsymbol{\sigma}_\phi(\mathbf{x})). \tag{2.46}$$

Using the recognition model the approximation of the likelihood function becomes the expected value of the conditional distribution with the latent variable distributed as the recognition model

$$p_\theta(\mathbf{x}^{(i)}) \approx \mathbb{E}_{\boldsymbol{z}\sim q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}[p_\theta(\mathbf{x}^{(i)}|\mathbf{z})]. \tag{2.47}$$

The relation between this approximation and the true likelihood function is a cornerstone in the VAE method [19]. To find this relation the concept of Kullback-Leibler divergence (see section 2.4.1) and variational lower bound is used.

2.5.1 Variational Lower Bound

For the approximation of the posterior distribution to be as accurate as possible, the recognition model $q_\phi(\mathbf{z}|\mathbf{x})$ should be as close as possible to the true posterior $p(\mathbf{z}|\mathbf{x})$. This performance is measured using the Kullback-Leibler divergence $D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}|\mathbf{x}))$. Using some algebraic manipulation and Bayes theorem the equation can be used to relate the true and approximated likelihood functions

$$\begin{aligned}
D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}|\mathbf{x})) = \\
\mathbb{E}_{z\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] = \\
\mathbb{E}_{z\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log q_\phi(\mathbf{z}|\mathbf{x}) - \log p(\mathbf{z}|\mathbf{x})\right] = \\
\mathbb{E}_{z\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log q_\phi(\mathbf{z}|\mathbf{x}) - \log \frac{p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p_\theta(\mathbf{x})}\right] = \\
\mathbb{E}_{z\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log q_\phi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{x}|\mathbf{z}) - \log p(\mathbf{z}) + \log(p_\theta(\mathbf{x})\right].
\end{aligned} \tag{2.48}$$

The third equality is given by applying Bayes theorem and by using the equality of the first and last expression the equality

$$\log p_\theta(\mathbf{x}) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}|\mathbf{x})) = \mathbb{E}_{z\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x}) + \log p(\mathbf{z})\right] \tag{2.49}$$

is given. Here $p_\theta(\mathbf{x})$ does not depend on $\mathbf{z}$ and can then be moved outside the expectation. $p(\mathbf{z})$ is the prior distribution of the latent variable, which usually is chosen to be a multivariate spherical Gaussian $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. Because of the non-negative property of the Kullback-Leibler divergence the equation can be transformed to the inequality

$$\log p_\theta(\mathbf{x}) \geq \mathbb{E}_{z\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) = \mathcal{L}(\mathbf{x}; \boldsymbol{\theta}, \boldsymbol{\phi}). \tag{2.50}$$

Where $\mathcal{L}(\mathbf{x};\boldsymbol{\theta},\boldsymbol{\phi})$ is a lower bound of the log-likelihood function $\log p_\theta(\mathbf{x})$ and is called the lower variational bound, which relates the true likelihood with the posterior approximation. Because the log-likelihood function is intractable to compute and find the maximum, the lower variational bound can be used for the optimization instead [12]

$$\arg\max_{\boldsymbol{\theta}} p_\theta(\mathbf{x}) \approx \arg\max_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathcal{L}(\mathbf{x};\boldsymbol{\theta},\boldsymbol{\phi}). \tag{2.51}$$

2.5.2 Optimization

To optimize the lower variational bound, stochastic gradient descent can be used, the method is explained in section 2.1.4. The gradient of the objective function $\mathcal{L}(\mathbf{x};\boldsymbol{\theta},\boldsymbol{\phi})$ must then be computed. To achieve this a closed form expression is needed for the objective function. When the recognition model is as a Gaussian distribution (equation 2.46) and the prior distribution is a spherical Gaussian, this can be found for the Kullback-Leibler divergence term and is shown in appendix A.1. But for the term of the expected value a closed form expression is not possible to find, instead it can be estimated by sampling of the latent variable

$$\mathbb{E}_{z\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] \approx \frac{1}{L}\sum_{i=1}^{L} \log p_\theta(\mathbf{x}|\mathbf{z}^{(i)}), \quad \mathbf{z}^{(i)} \sim q_\phi(\mathbf{z}|\mathbf{x}). \tag{2.52}$$

The gradient operator can be applied to this estimated function and is given by

$$\nabla_{\theta,\phi}\mathcal{L}(\mathbf{x};\boldsymbol{\theta},\boldsymbol{\phi}) \approx \nabla_{\theta,\phi}\left(\frac{1}{L}\sum_{i=1}^{L}\log p_\theta(\mathbf{x}|\mathbf{z}^{(i)}) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))\right), \quad \mathbf{z}^{(i)} \sim q_\phi(\mathbf{z}|\mathbf{x}) \tag{2.53}$$

But the sampling of the latent variables $\mathbf{z}^{(i)}$ causes problems with the gradient of the parameters $\boldsymbol{\phi}$ of the recognition model. The sampling is a non-continuous operation which has no gradient, the error from the likelihood function $\log p_\theta(\mathbf{x}|\mathbf{z}^{(i)})$ can then not be propagated through this operation and be applied to the function $q_\phi(\mathbf{z}|\mathbf{x})$. To get around this problem the reparametrization trick can be used.

2.5.3 Reparameterization Trick

Instead of sampling from the Gaussian distribution $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z};\boldsymbol{\mu}_\phi(z|x),\boldsymbol{\sigma}_\phi(z|x))$, the sampling can be reparametrized by using a auxiliary variable $\boldsymbol{\epsilon}$. This transforms the sampling of $\mathbf{z}^{(i)}$ to a deterministic function to which the gradient operator can be applied

$$\mathbf{z}^{(i)} = g_{\boldsymbol{\phi}}(\mathbf{x},\boldsymbol{\epsilon}) = \boldsymbol{\mu}_\phi(\mathbf{x}) + \boldsymbol{\sigma}_\phi(\mathbf{x})\boldsymbol{\epsilon}, \quad \epsilon \sim \mathcal{N}(\mathbf{0},\mathbf{I}) \tag{2.54}$$

Which is continuous and possible to apply a gradient operator to.

The VAE model can be visualized for better understanding and is shown in figure 2.8. Summarizing the method, it consists of approximating the recognition model $q_\phi(\mathbf{z}|\mathbf{x})$ and conditional distribution $p_\theta(\mathbf{x}|\mathbf{z})$, by optimizing the lower variational bound $\mathcal{L}(\mathbf{x};\boldsymbol{\theta},\boldsymbol{\phi})$. This is done using the data points $\mathbf{x}^{(i)}$, sampling of auxiliary variables $\boldsymbol{\epsilon}$ and updating the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$. Gradient descent is used as the method for updating the parameters and is given by

$$[\Delta\boldsymbol{\theta}, \Delta\boldsymbol{\phi}] = \nabla_{\theta,\phi}\left(\frac{1}{M}\sum_{i=1}^{M}\left(\frac{1}{L}\sum_{l=1}^{L}\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)})\right) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p(\mathbf{z}))\right),$$
$$\mathbf{z}^{(i,l)} = \boldsymbol{\mu}_\phi(\mathbf{x}^{(i)}) + \boldsymbol{\sigma}_\phi(\mathbf{x}^{(i)})\boldsymbol{\epsilon}^{(i,l)}, \quad \boldsymbol{\epsilon}^{(i,l)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}). \tag{2.55}$$

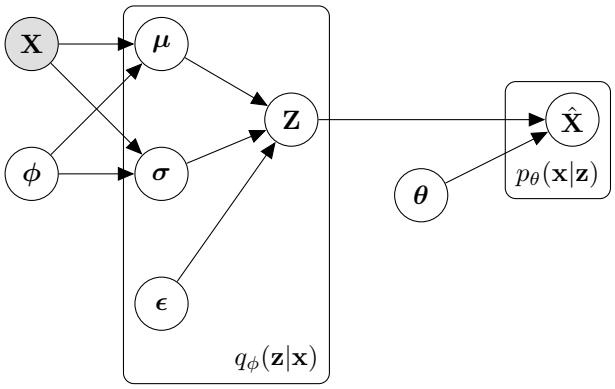


Figure 2.8: VAE model visualized by a Bayesian plate notation. The input data $\mathbf{X}$ and the parameters $\boldsymbol{\theta}$ defines the approximation of the posterior distribution of the latent variable $\mathbf{Z}$ by the recognition model $q_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$. The latent variable and the parameters $\boldsymbol{\theta}$ then defines the conditional distribution $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$

2.5.4 Implementation

A natural choice to implement the recognition model $q_\phi(\mathbf{z}|\mathbf{x})$ and the conditional distribution $p_\theta(\mathbf{x}|\mathbf{z})$ is by neural networks. The recognition model is implemented by multiple layers mapping the input $\mathbf{x}$ to the vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ defining the latent variable $\mathbf{z}$. This part of the network is usually called the encoder. The conditional distribution is implemented by layers mapping the latent variable $\mathbf{z}$ to a generated vector of the same size as the input variable $\mathbf{x}$. This part of the network is called decoder or generator.

2.6 Generative Adversarial Networks

Generative Adversarial Networks (GAN) were introduced in [11] and have since received much attention, in particular for their ability to generate believable images. GANs learn an approximate mapping from a simple distribution to a target distribution p_{data}, generally a dataset, allowing sampling of that approximation. GANs are trained by an adversarial process in which two models are trained simultaneously: a Generator G which generates data as similar as possible to the data distribution and a discriminator D which classifies data as coming from the real data distribution or from G. The generator, G, is trained by maximizing the probability of D miss-classifying a generated sample as real.

More formally, given a data space $\mathcal{X}$ and a latent space $\mathcal{Z}$, D is a mapping $\mathcal{X} \to [0, 1]$, which is trained to classify data as real or fabricated. G is a mapping $\mathcal{Z} \to \mathcal{X}$, i.e. from the latent space to data space, trained to maximize the probability of D making an error.

The models are jointly optimized by playing the minmax game:

$$\min_G \max_D V(D, G), \tag{2.56}$$

where V(D,G) is the loss function:

$$V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})}[\log(D(\boldsymbol{x}))] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]. \tag{2.57}$$

That is, the loss function is minimized by the generator and maximized by the discriminator. This minmax game has a global optimum when $p_{data} = p_{model}$, i.e. when the generated distribution exactly matches the data distribution. For an optimal discriminator, it can be shown that minimizing $C(G) = \max_D V(D, G)$ corresponds to minimizing the Jensen-Shannon divergence between the two distributions [11].

In practice both D and G are generally neural networks, in this case the entire model can be trained through backpropagation with loss function $V(D, G)$. This is then done by alternating between two steps:

- **Discriminator Training:** Samples are drawn from $\boldsymbol{z} \sim \mathcal{U}(0, 1)$ and p_{data}, both data samples and $G(\boldsymbol{z})$ are fed through D. The loss $\mathbb{E}_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})}[\log(D(\boldsymbol{x}))] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$ is calculated, the gradients are backpropagated and the weights of D are updated.

- **Generator Training:** Samples from $\boldsymbol{z} \sim \mathcal{U}(0, 1)$ are fed through G and $G(\boldsymbol{z})$ is fed through D. The loss $\mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$ is calculated, the gradients are backpropagated and the weights of G are updated.

To ensure that the gradients propagated back to the generator are good, the discriminator training step is often performed 1-5 times for each step of generator training.

GAN training with the training objective in eq 2.57 is generally not effective in practice. The reason for this is that samples from G are often rejected with high confidence by D. These rejections happen in the saturating region of the cross-entropy, resulting in vanishing gradients for the generator. A non-saturating heuristic is often used in place of the standard loss function. Instead of minimizing $\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$ the generator maximizes $\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(D(G(\boldsymbol{z})))]$. This yields a strong gradient signal to the generator, in particular when the discriminator makes confident predictions. Note that the discriminators loss function is unchanged.

Training of Generative adversarial networks is not guaranteed to converge and suffers from instability. A common problem is that of mode collapse, where the generator learns to generate data from only a fraction of the modes present in the data and hence does not cover the data distribution well. Other known failure modes are vanishing gradients and indefinitely oscillating non-convergent behaviour.

Several approaches [3], [20], [21], [22], [23], [24]. have been suggested to improve stability and address the issue of mode collapse. While much progress has been made, the majority of solutions are heuristically motivated and come with severe restrictions on the architectures used. In this thesis, the issue will be addressed by using the modified GAN formulation proposed in *Wasserstein GAN* [20], in combination with the heuristics proposed in *Deep Convolutional Generative Adversarial Networks* [21].

2.6.1 Bidirectional Generative Adversarial Networks

Bidirectional Generative Adversarial Networks(BiGAN) [5] is an extension to the GAN framework which in addition to D and G aims to learn an Encoder E, which is an approximate inverse of G.

To train E, the discriminator takes as input both latent vectors and data. D is fed either a latent vector and the corresponding generated image $[G(\boldsymbol{z}), \boldsymbol{z}]$ or an image and the corresponding encoded latent vector $[\boldsymbol{x}, E(\boldsymbol{x})]$. D is then tasked to distinguish between the two. The modified loss function is:

$$V(D, G, E) = \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log(D(\boldsymbol{x}, E(\boldsymbol{x}))] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}), \boldsymbol{z}))], \quad (2.58)$$

where the three networks are optimized with the minmax objective

$$\min_{G,E} \max_{D} V(D, G, E).$$

If the discriminator has enough capacity, it should hence learn the correspondence between latent space and data, by extension this means that E has to invert G in order to generate samples which yield a low output from D.

Similarly to GAN, neural networks can be used to model E,G and D The entire model can then be trained through back-propagation. The training is done in by alternating between two steps:

- **Discriminator training:** Data is sampled from both p_{data} and $\boldsymbol{z} \sim \mathcal{U}(0,1)$ and fed through the encoder, generator and discriminator. The loss $V(D,G,E)$ is calculated, gradients are backpropagated and the weights of D are updated in the positive gradient directions.

- **Generator/Encoder training:** Data is sampled from both p_{data} and $\boldsymbol{z} \sim \mathcal{U}(0,1)$ and fed through the encoder, generator and discriminator. The loss $V(D,G,E)$ is calculated, gradients are backpropagated and the weights of G and E are updated in the negative gradient direction.

To avoid the problem of vanishing gradients the reversed generator loss function often used in GAN is here used for both the encoder and generator. Instead of minimizing $V(D,G,E)$ from eq 2.58, the encoder and generator maximize

$$\tilde{V}(D,G,E) = \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log(1 - D(\boldsymbol{x}, E(\boldsymbol{x}))] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(D(G(\boldsymbol{z}), \boldsymbol{z}))]. \quad (2.59)$$

The encoding learned by E through this procedure should capture important features of the data and could hence be employed to learn features from unannotated data.

Another method for inverting the generator is using a *latent regressor* [5]. The Encoder is simply trained to reconstruct the input of the Generator given its output. The latent regressor loss function hence is the reconstruction loss

$$L(E) = ||E(G(z)) - z||^2, \quad (2.60)$$

similar to an autoencoder. The encoder can be trained in two ways: **jointly** with the generator during GAN training, or **separately**, in this case a generator is first trained to convergence in a GAN setup, the encoder is then trained with the generator weights fixed.

2.6.2 Deep Convolutional Generative Adversarial Networks

DCGAN [21] extends GANs to deep convolutional neural networks and suggests architecture restrictions which stabilize training. The architecture proposals are:

- strided convolution instead of pooling and fractionally strided convolution instead of unpooling
- batch normalization in both generator and discriminator
- rectified activations and tanh output activation in generator
- leaky rectified activations in discriminator
- remove fully connected layers from deep architectures

A simple CNN architecture is proposed and empirically shown to be stable on a number of tasks, this architecture is used as the starting point of all convolutional models used in combination with GAN in this thesis.

2.6.3 Wasserstein Generative Adversarial Networks

Wasserstein Generative Adversarial Networks(WGAN) [20] are a recently proposed modification to GAN. In [25], [20] the stability issues of GAN are studied in depth, it is argued that the minimization of the Jensen-Shannon divergence is inherently problematic since it may not be continuous if p_{data} and p_{model} both lay on manifolds of non-full dimension and are not perfectly aligned. The Wasserstein distance, or Earth-Mover distance, is proposed as an alternative. The Wasserstein distance is defined as:

$$W(p_{data}, p_{model}) = \inf_{\gamma \in \Pi(p_{data}, p_{model})} \mathbb{E}_{(x,y)\sim\gamma} \left[||x - y|| \right], \tag{2.61}$$

where $\Pi(p_{data}, p_{model})$ denotes the sets of all joint distributions (x, y) with marginal distributions p_{model} and p_{data}. A more intuitive explanation of the Wasserstein distance is that it corresponds to the probability mass times distance required to move one distribution into the other given an optimal transport plan. Since the infimum of the Wasserstein distance makes it intractable to calculate in practice, it can instead be approximated as

$$\max_{w \in \mathcal{W}} \mathbb{E}_{x\sim p_{data}}[D_w(x)] - \mathbb{E}_{z\sim p_z}[D_w(G_\theta(x))], \tag{2.62}$$

where D_w is a K-Lipschitz function parametrized by w and G_θ is a function parametrized by θ. In practice this can be achieved by modelling D_w and G_θ as neural networks, and the K-Lipschitz constraint can be fulfilled by bounding the weights of D to some constant after every training iteration. Wasserstein GAN alleviates many of the known issues of GAN, in particular mode collapse and vanishing gradients. In addition, the Wasserstein distance correlates well with sample quality and can be used as an indicator for the training progress, something the standard GAN loss does not provide. One backside of this method is that the weight constraints imposed on the discriminator bias it towards simpler functions [24] and restricts the discriminators capacity.

In [24] a more efficient way of imposing the K-Lipschitz constraint on the discriminator is proposed, one which does not bias the discriminator towards simpler functions.

A function is K-Lipschitz exactly if it has gradients of norm K or less everywhere. Instead of enforcing this by bounding the weights of D, a soft constraint which directly encourages the discriminator to have norm 1 gradients is proposed. The method consist of simply adding a gradient penalty term to the discriminator loss, the term is defined as:

$$\lambda \mathbb{E}_{\hat{x}\sim P_{\hat{x}}} \left[\left(||\nabla_{\hat{x}} D(\hat{x})|| - 1 \right)^2 \right], \tag{2.63}$$

where the points $\hat{x}$ are sampled along lines between generated and true data samples. The gradient penalty is then added to the Wasserstein GAN loss function as:

$$L = \mathbb{E}_{x\sim p_{data}}[D_w(x)] - \mathbb{E}_{z\sim p_z}[D_w(G_\theta(x))] + \lambda \mathbb{E}_{\hat{x}\sim P_{\hat{x}}} \left[\left(||\nabla_{\hat{x}} D(\hat{x})|| - 1 \right)^2 \right] \tag{2.64}$$

This method of GAN training is shown to be robust to choice of architecture and produce high quality samples. This formulation of GAN will be referred to as to as Wasserstein GAN with gradient penalty or WGAN-GP.

2.7 Feature Learning with Auxiliary tasks

Feature learning from auxiliary tasks is an example of a set of techniques denoted self-supervised learning. The idea is to perform supervised learning, but not on the intended task, instead the model is trained on similar task where ground truth data is available. Generally this involves using the data itself as ground truth. One examples of feature learning using an auxiliary task is *learning by inpainting* [26], where parts of images are removed and the model is tasked with painting in the missing parts. This way, the data itself can be used as ground truth. The basic idea behind training with auxiliary tasks that a feature representation that is effective for one task is likely to be effective for other similar tasks.

The auxiliary task studied in this thesis is predicting the next image in a sequence of images, this allows for leveraging the temporal aspect of the data to train a feature extractor. For instance predicting the position of objects in an image from a short sequence of preceding images would require a feature representation which distinguished between static and dynamic object, such a feature representation could then be useful for other tasks where that information is relevant.

3

Method

This chapter describes and explains how the theory presented in chapter 2 is implemented and applied to the problem of semantic segmentation studied. First, the main task of semantic segmentation is described. The data which was used is then presented, along with details of how it was processed and used in the experiments. Then, the implementation of the different learning methods used in this thesis is described. In addition, details of how the methods were designed, implemented and evaluated are covered. This chapter also presents a previously developed method for the problem, which is used as a baseline comparison for the methods in this thesis.

3.1 Semantic segmentation

In this thesis, the main problem that has been studied is the task of performing semantic segmentation, a supervised learning task generally applied to image data. The goal when performing semantic segmentation is to partition an image into parts which are semantically meaningful. More formally, the studied task is to perform pixel-wise class prediction, i.e., assigning each pixel in the input image to a class, from a set of predefined classes. This means that the number of output variables of a semantic segmentation model is equal to the number of input pixels. In other words, semantic segmentation is a classification task with as many response variables as input pixels. This thesis focuses on road segmentation, that is, labeling pixels in an image as road or non-road.

3.2 Datasets

Two datasets were used to evaluate the performance of feature learning with GAN and VAE, namely the MNIST handwritten digits dataset and the KITTI road segmentation dataset. The handwritten digits served as an easily analysed problem of low complexity for proof of concept, while feature learning on the KITTI data was the main task. On the kitti dataset, feature learning through an auxiliary task was also evaluated.

3.2.1 MNIST

The MNIST handwritten digit database is commonly used as a basic classification problem to evaluate machine learning methods. The data set consists of 60000 images of handwritten digits in 28×28 resolution, divided into 50000 training examples and 10000 test examples, as well as corresponding labels for each digit. The digits are size normalized and center cropped and the dataset hence requires minimal pre-processing.

To evaluate the studied feature learning methods on a simple semantic segmentation task, a slightly modified MNIST dataset was created. Instead of using the labels directly, a semantic segmentation problem was created by recasting the labels as pixel-wise labels. This was done by thresholding the images yielding binary images divided into background and digit. The pixels corresponding to digit were assigned the digit label, while the pixels corresponding to background were assigned to a background class. When training neural networks on this semantic segmentation task, only 500 samples were used for training the model, while the remainder were used to validate the performance on previously unseen data.

3.2.2 KITTI

The KITTI dataset [27] was used for training and evaluating the investigated methods. The dataset consists of images, lidar scans and other measurements from sensors mounted on a car. There are 47885 lidar-image pairs from 156 separate sequences, 289 out of these images are densely annotated (pixel-by-pixel) into the classes of *road* and *not road* shown in figure 3.1. The images and annotations are stored as RGB and have a resolution of 370x1226 pixels. The lidar data is represented as point clouds, arrays of approximately 120000 values per measurement, each consisting of a x,y,z coordinate and a reflectance value.

To get a data representation which is compatible with common neural network architectures, the lidar points were projected into a bird's-eye view perspective. The method is based on the work in [9]. A window ranging from 6 to 46 meters in front of the car and 10 meters to each side was considered. The window was binned into 0.1x0.1 m cells, the lidar points were then allocated into these bins. In each bin various statistics were calculated, namely:

- min height
- mean height
- max height
- standard deviation of height
- point density (along z-axis)
- min reflectance
- mean reflectance

- max reflectance
- standard deviation of reflectance
- density (along reflectance)
- number of points

Each of these statistics were then considered a feature map, meaning the each lidar scan was represented as an $400 \times 200 \times 11$ matrix. To keep the computational complexity down, only the mean height channel was used for training the generative models.

To be able to utilize the annotations provided with the kitti dataset, they were projected into the same bird's-eye view perspective as the lidar points. An example of the image and annotation format can be seen in figure 3.1 and an example of the bird's-eye view perspective and lidar points can be seen in figure 3.2. The kitti dataset provides projection matrices between the velodyne lidar sensor and the camera coordinate systems. In combination with a projection from the image plane to the bird's-eye view perspective, the road annotations can be used for the lidar data. In figure 3.2 an illustration of the transformations are shown.

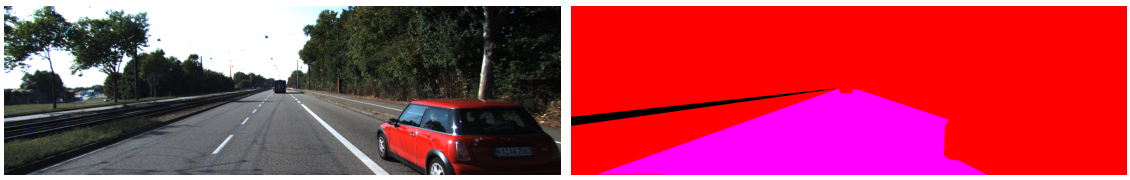


Figure 3.1: Example of an image from KITTI data with the corresponding annotations into the class road (pink) and not road (red).

In all the models trained for road segmentation in this thesis, the annotated data was split into a training set and a validation set. The training set was used to train the model, while the validation set was used to validate the performance of the neural networks for road segmentation on data not previously seen by the model. 259 of the annotations were used for training and the remaining 30 were used for validation.

The raw KITTI data consists of sequences of camera and lidar images captured from a moving vehicle, and are hence defined in a moving reference frame. When using several frames simultaneously, such as in the method presented in section 3.3.2, the frames should ideally be expressed in the same reference frame. The KITTI dataset provides measurements of velocity and angular momentum of the vehicle for every captured frame. This information was used to transform sets of three frames to a common reference frame, by transforming the raw lidar measurements before they were projected into the bird's-eye view.

The measurements in a frame at time t are given by $\mathbf{p}_t$. These measurements can be transformed to be represented in a reference frame at time t' by

$$\tilde{\boldsymbol{p}}_t = (R \cdot \boldsymbol{p}_t + T). \tag{3.1}$$

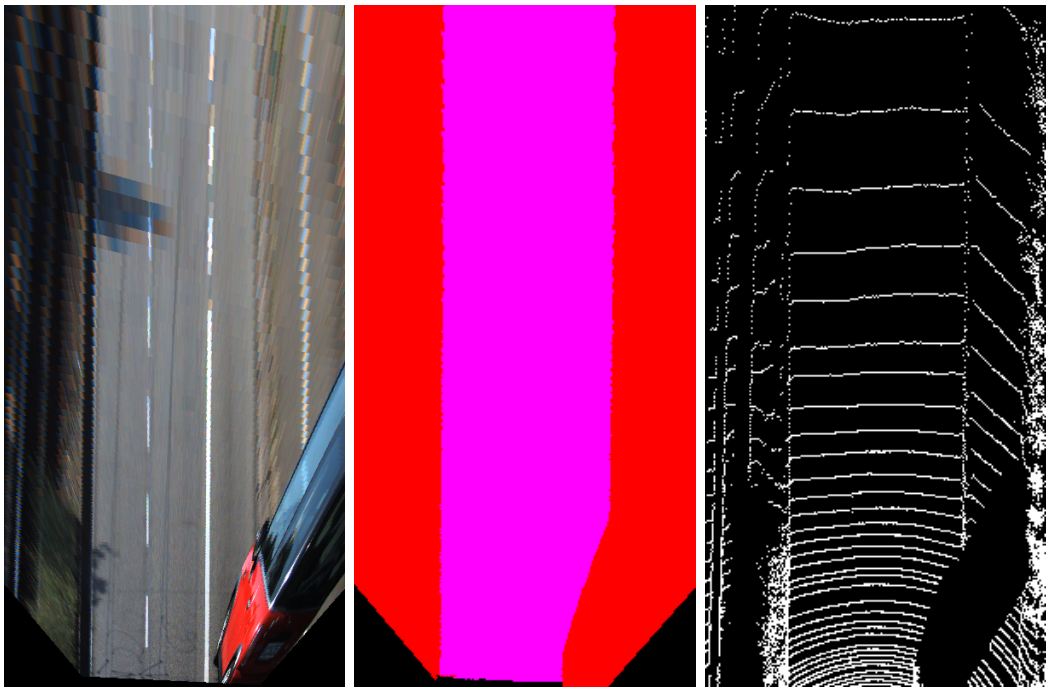


Figure 3.2: Left figure shows the image from figure 3.1 transformed to the bird's-eye view perspective. Center images shows the corresponding annotations and the right figure shows the lidar measurements projected to the same bird's-eye view.

Where R is the rotation matrix

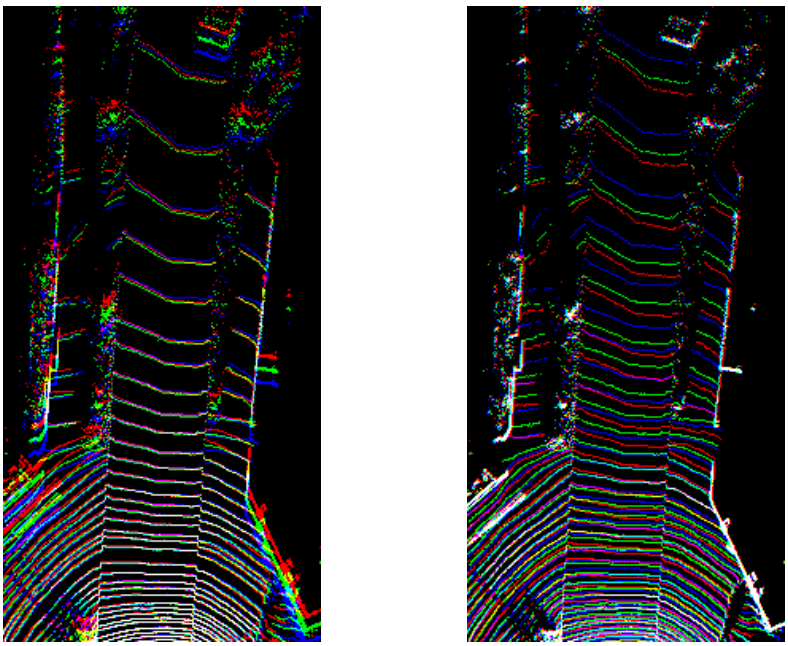
$$R = \begin{pmatrix} \cos\theta^{[t]} & -\sin\theta^{[t]} \\ \sin\theta^{[t]} & \cos\theta^{[t]} \end{pmatrix} \tag{3.2}$$

and T is the translation vector

$$T = (t' - t) \begin{pmatrix} v_x^{[t]} \\ v_y^{[t]} \end{pmatrix}. \tag{3.3}$$

Here $\theta^{[t]} = (t' - t)\gamma^{[t]}$ is the angular difference between the current frame and the reference frame. The difference is determined by $\gamma^{[t]}$ which is the angular velocity around the z-axis at time t. $v_x^{[t]}$,$v_y^{[t]}$ denotes the velocity in the current frame in the x and y direction respectively.

The transformations are applied to the set of three frames at times $[t-1, t, t+1]$. The previous frame at $t-1$ and next frame at $t+1$ are transformed to be represented in the reference frame at time t. In figure 3.3 this transformation is illustrated.



(a) Original **(b)** Aligned

Figure 3.3: Bird's-eye view of lidar points before and after compensation for ego-motion. The red channel contains points detected at $t = -0.1s$, the green channel contains detections at $t = 0$ and the red channel contains detections from $t = 0.1s$.

3.3 Feature Learning

Evaluating the feasibility of learning features through generative models and auxiliary tasks is the main focus of this thesis. To this end, two data sets (MNIST and KITTI) were utilized. In both cases, a large amount of unannotated data was used in unsupervised feature learning and a small set of annotated data was used to train models for semantic segmentation. Two generative models and their application towards unsupervised feature learning were studied on both MNIST and KITTI data, namely GAN and VAE. In addition, the process of learning features by solving an auxiliary task was studied on the KITTI dataset.

3.3.1 Generative Models

When using generative models to learn features for semantic segmentation, the process can be divided into a **Generative stage** and a **Supervised Stage**, which are outlined as follows:

- **Generative stage:** A generative model is trained using unlabeled data. By utilizing the variational lower bound in the case of VAE and the discriminators adversarial loss in the case of GAN, the models are trained until convergence of the loss measures. The encoder of the models should then give an efficient feature representation by mapping the input data to the latent variable. The parameters of this encoder are stored for use in the supervised stage.

- **Supervised stage:** An encoder decoder network is created for a semantic segmentation task. The training of this network is performed using annotated data and pixel-wise cross entropy loss. The encoder weights are initialized to pre-trained weights from a generative model, and the decoder is initialized randomly. The decoder is then trained to convergence while the encoder remains fixed. Optionally, the entire network can then be trained, a step referred to as fine tuning. The idea is that the encoder features, trained on a significantly larger dataset, should improve model generalization and speed up convergence.

The studied generative models were implemented on a variety of architectures and evaluated quantitatively in the generative stage. The models were evaluated on generated sample quality, reconstruction quality and in the case of GAN, training stability. The models which yielded promising results were then evaluated in the supervised stage.

3.3.2 Next-frame Prediction

The auxiliary task of sequence prediction was studied as a feature learning strategy for the road segmentation task. More specifically, the studied task was predicting the

next lidar image, given a sequence of lidar images. The data representation covered in section 3.2.2 was utilized. Intuitively, this task should require a feature representation that distinguishes between static and dynamic objects, as well as between flat objects(road/sidewalk) and tall objects(buildings/vegetation), by extension such a representation should also be useful for road segmentation.

The suggested approach is to feed a neural network with a sequence of images as input. The output of the network should then be the next image in the sequence. A sequence of length 3 was used. The previous image at time $n-1$ and the present image at time n was used as input. The network was then tasked to predict the next image in the sequence, that is, the frame at time $n+1$. Since the data representation is sparse, and prediction should be dense, only the pixels which contain a lidar detection were used as ground truth, the loss functions hence takes the form:

$$L = \sum_{\forall i,j:\boldsymbol{x}_{ij}\neq 0} ||\tilde{\boldsymbol{x}}_{ij} - \boldsymbol{x}_{ij}||_2^2. \tag{3.4}$$

This means the task is not to predict where the lidar detections land but rather the underlying state (actual ground height), which was then only updated where measurements are available. The model hence outputs a dense height prediction even though trained with sparse labeling.

Two different procedures for using this auxiliary task for feature learning were evaluated. The first, referred to as *separate training*, consists of first training a neural network to convergence on the next-frame prediction task and then utilizing the trained weights as initialization for training the network on the road segmentation task. The other procedure is referred to as *joint training*. In this case, the two tasks are trained jointly sharing an encoder network, but with two separate decoder networks. In this setting the auxiliary task acts as a regularizer, biasing the encoder training towards features which are useful for both tasks, which should benefit model generalization.

3.4 Model Evaluation

Two semantic segmentation tasks were chosen to evaluate the feature learning approach, the constructed handwritten digit segmentation task described in section 3.2.1, and road segmentation on the KITTI road dataset from section 3.2.2. Several feature learning models were evaluated in combination with several neural network architectures.

In all experiments a supervised baseline was established by training the network from randomly initialized weights, and evaluating its performance on validation data. The features learned from generative models were then assessed on the improvement in performance and convergence rate over this baseline.

3.4.1 Evaluation metrics

The performance of the trained models was measured by a number of different metrics. Using the number of correctly and falsely classified pixels different metrics were computed. True positives (TP), true negatives (TN), false negatives (FN) and false positives (FP) were used to compute the accuracy (acc), precision, recall, F1-score, Fmax-score and Intersection over Union (IoU) [27].

The accuracy is a metric for the fraction of the total number of pixels classified correctly. For a class i it is given by

$$Acc = \frac{TP_i + TN_i}{TP_i + FN_i + FP_i + TN_i}. \tag{3.5}$$

Precision measures how precise the classification of a class is. The metric is the fraction of correctly classified pixels over the total number of pixels classified to a class and is defined by:

$$precision = \frac{TP_i}{TP_i + FP_i}. \tag{3.6}$$

Recall is a measure of what fraction of a class was correctly classified. It is given as the fraction of correctly classified pixels over the total number of pixels of the class and is defined as:

$$recall = \frac{TP_i}{TP_i + FN_i}. \tag{3.7}$$

Precision and recall are unbalanced metrics by themselves. For example classifying all pixel to a certain class will give perfect recall but can give a low precision. So a more informative metric are given by combining both metrics using the F_1-score, which is the harmonic mean of the two metrics and is defined as:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}. \tag{3.8}$$

This metric can also be extended to the F_{max}-score, where the optimal classification threshold τ is used. This threshold is the value which the softmax output of the class needs to exceed to make the class prediction. The metric is defined as:

$$F_{max} = \max_{\tau} 2 \cdot \frac{precision(\tau) \cdot recall(\tau)}{precision(\tau) + recall(\tau)}. \tag{3.9}$$

Another metric used to measure the performance of the pixel classifications is IoU. It is accuracy measure for the positive classifications of class and given by:

$$IoU = \frac{TP_i}{TP_i + FN_i + FP_i}. \tag{3.10}$$

By taking the mean over all classes the mean intersection over union is obtained by

$$mIoU = \frac{1}{N} \sum_{i \in \mathcal{C}} \frac{TP_i}{TP_i + FN_i + FP_i}. \tag{3.11}$$

where $\mathcal{C}$ is the set of all N classes.

3.5 Handwritten digit segmentation

The task of semantically segmenting handwritten digits, described in section 3.2.1, was used as a proof of concept problem to evaluate the studied unsupervised feature learning methods. To create a challenging task, where unlabeled data could be leveraged, only a small subset of the MNIST digits (500 samples) were used for supervised training. A simple CNN architecture was used for all experiments.

A supervised baseline was established through training the model fully supervised from randomly initialized weights. The encoder was then trained in various generative setups, both as an encoder in a VAE and a latent regressor in a GAN.

3.5.1 Network architecture

The network used for the MNIST experiments was a simple encoder decoder architecture, which can be seen in figure 3.4. The network architecture is summarized in table 3.1. All layers except the output layer are followed by batch normalization and a ReLU activation, the output layer is a 11-way softmax function. Half padding is used everywhere and a dropout rate of 25% is used in the two first decoder layers.

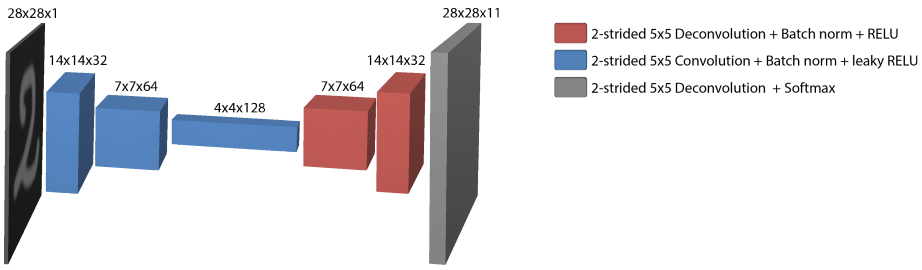


Figure 3.4: A visualization of the network architecture used for semantic segmentation of MNIST digits in the supervised stage.

Layer	
1	5x5 conv, stride 2 (14x14x32)
2	5x5 conv, stride 2 (7x7x64)
3	5x5 conv, stride 2 (4x4x128)
4	5x5 deconv, stride 2 (7x7x64)
5	5x5 deconv, stride 2 (14x14x32)
6	5x5 deconv, stride 2 (28x28x1)

Table 3.1: Description of the CNN used in the supervised stage of semantic segmentation of MNIST.

In the generative stage, the first three layers of the network depicted in figure 3.4 were trained as an encoder in both GAN and VAE models. When trained as part of a GAN with a latent regressor, the encoder was paired with a discriminator and

a generator. The neural network architectures of all three models are summarized in table 3.2. All intermediate layers use ReLU activations. The generator used a tanh output, while both encoder and discriminator use linear outputs. Batch normalization was utilized between all layers in the encoder and generator, while layer normalization was used between all discriminator layers.

In the VAE setting the encoder part of the model consists of the first three layers of the architecture and an additional layer added for mapping to the latent space. This additional layer consists of two parallel dense layers mapping the 4x4x128 feature volume to a vector of mean values ($\boldsymbol{\mu}$) and standard deviation values ($\boldsymbol{\sigma}$), both with the size of the latent dimension. The decoder part of the model takes an input from the distribution of the latent variable given by $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. The input is fed through an initial dense layer and is reshaped to the feature volume of size 4x4x128. The decoder then consists of the first two deconvolution layers shown in figure 3.4. The output layer was replaced by a deconvolutional layer with kernel size 5, stride 2 and a sigmoid activation function.

Layer	Encoder	Generator	Discriminator
1	5x5 conv, stride 2 (14x14x32)	dense + reshape (4x4x128)	5x5 conv, stride 2 (14x14x32)
2	5x5 conv, stride 2 (7x7x64)	5x5 deconv, stride 2 (7x7x64)	5x5 conv, stride 2 (7x7x64)
3	5x5 conv, stride 2 (4x4x128)	5x5 deconv, stride 2 (14x14x32)	5x5 conv, stride 2 (4x4x64)
4	dense (100)	5x5 deconv, stride 2 (28x28x1)	dense (1)

Table 3.2: Description of encoder, generator and discriminator used in the GAN model used in the generative stage of MNIST segmentation.

After the generative models were trained a supervised semantic segmentation task was performed. In this supervised stage the model was trained with 500 labelled examples, using pixel wise cross-entropy loss. The encoder was initialized with the weights learned in the generative stage. The decoder was then trained to convergence (minimum validation loss), the performance at this stage is referred to as fixed-encoder performance. Once the decoder had converged, both encoder and decoder were trained jointly until convergence, the performance at this stage is referred to as fine-tuned performance.

3.6 Road segmentation

Road segmentation on the KITTI road dataset was chosen as a relevant and larger-scale problem to assess whether or not the studied methods scale to higher resolution data and more complex neural network architectures. In addition, the availability of a large amount of unlabeled sequence data made the dataset suitable for this study.

Two different network architectures were implemented to evaluate the feature learning methods, both architectures are encoder-decoder style fully convolutional networks. One architecture closely resembles SegNet [28], which generally performs well on semantic segmentation tasks. The other architecture is the architecture from [9], which is specifically designed for road segmentation from lidar images.

In the generative stage, the encoder part of these networks were trained in combination with suitable generator and discriminator architectures (described in section 3.6.1), in the GAN framework, and a suitable decoder in the VAE framework.

3.6.1 SegNet style architecture

The SegNet architecture is an encoder-decoder style network for semantic segmentation. It was introduced in [10] and has shown promising results for semantic segmentation task on various different data sets. The network consists of a series of convolutions and max pooling layers for down sampling followed by deconvolutions and unpooling layers for upsampling. The design of the network is based on the widely known VGG16 network [29], which has been used as a benchmark for image classification tasks. By using the same design in the encoder part of the SegNet, it is simple to use weights from Imagenet pre-training in the the encoder part of the network. The purpose of studying the SegNet architecture is that it provides a simple way to compare Imagenet pre-training to unsupervised feature learning. In table 3.3 the design of the slightly modified SegNet architecture used in this thesis is shown. Every convolutional layer is followed by a ReLU activation, and dropout was applied after every pooling layer.

Encoder	**Decoder**
3x3 conv, stride 1 (400x200x32)	3x3 deconv, stride 1 (25x13x512)
3x3 conv, stride 1 (400x200x32)	3x3 deconv, stride 1 (25x13x512)
2x2 max pool (200x100x32)	3x3 deconv, stride 1 (25x13x512)
3x3 conv, stride 1 (200x100x64)	2x2 max unpool (50x25x512)
3x3 conv, stride 1 (200x100x64)	3x3 deconv, stride 1 (50x25x256)
2x2 max pool (100x50x64)	3x3 deconv, stride 1 (50x25x256)
3x3 conv, stride 1 (100x50x128)	3x3 deconv, stride 1 (50x25x256)
3x3 conv, stride 1 (100x50x128)	2x2 max unpool (100x50x256)
3x3 conv, stride 1 (100x50x128)	3x3 deconv, stride 1 (100x50x128)
2x2 max pool (50x25x128)	3x3 deconv, stride 1 (100x50x128)
3x3 conv, stride 1 (50x25x256)	3x3 deconv, stride 1 (100x50x128)
3x3 conv, stride 1 (50x25x256)	2x2 max unpool (200x100x128)
3x3 conv, stride 1 (50x25x256)	3x3 deconv, stride 1 (200x100x64)
2x2 max pool (25x13x256)	3x3 deconv, stride 1 (200x100x64)
3x3 conv, stride 1 (25x13x512)	2x2 max unpool (400x200x64)
3x3 conv, stride 1 (25x13x512)	3x3 deconv, stride 1 (400x200x32)
3x3 conv, stride 1 (25x13x512)	3x3 deconv, stride 1 (400x200x32)

Table 3.3: Description of SegNet style Encoder and Decoder used to evaluate performance of the features from generative models on the task of road segmentation. Each layer is described by the size of the kernel, the stride of the kernel and the output size of the layer.

3.6.2 DCGAN style network

When trained in the GAN framework, the encoder used for the supervised task was trained together with a discriminator and generator network. The generator and discriminator are based on the DCGAN architecture [21], both are extended

with extra layers to account for the larger image size used. Both architectures are described in table 3.3. The generator uses batch normalization and rectifier activations on all layers except the output, which uses no normalization and a tanh function. The discriminator uses layer normalization and leaky rectifier activations on all layers expect the output layer, which uses no normalization and no activation function.

Layer	**Generator**	**Discriminator**
1	dense + reshape (13x7x512)	5×5 conv, stride 2 (200x100x64)
3	5×5 deconv, stride 2 (25x13x512)	5×5 conv, stride 2 (100x50x128)
4	5×5 deconv, stride 2 (50x25x256)	5×5 conv, stride 2 (50x25x256)
5	5×5 deconv, stride 2 (100x50x128)	5×5 conv, stride 2 (25x13x512)
6	5×5 deconv, stride 2 (200x100x64)	5×5 conv, stride 2 (13x7x512)
7	5×5 deconv, stride 2 (400x200x1)	dense to single unit

Table 3.4: Description of generator and discriminator used to for unsupervised feature learning with GAN on KITTI road data. All generator layers except the last are followed by batch normalization and ReLU activations, the last layer is followed by a tanh activation. All discriminator layers except the last are followed by layer norm and leaky ReLU, while the last layer is linear.

3.6.3 ContextNet architecture

The architecture developed in [9], here denoted *ContextNet*, is specifically designed for the task of road segmentation from bird's-eye view lidar images. The architecture uses dilated convolutions to allow for large receptive fields without a prohibitive number of parameters and without significant downsampling. This allows the network to use global information for each pixel prediction, a property which is though to be important for road segmentation from the sparse lidar representation.

The architecture of the neural network can be seen in figure 3.5. The network consist of three major components: an encoder, a context module and a decoder. The encoder consists of two convolutional layers with unit stride and a kernel size of 3x3 and 32 channels followed by a 2x2 max pooling layer. The Context module consists of 7 3x3 dilated convolution layers with 128 channels followed by a 1x1 convolution layer with 32 channels. The dilation layers have exponentially growing dilation factors, which can be seen in table 3.5. The decoder consist of a max unpool operation followed by two 3x3 convolutional layers, the first with 32 channels and the second with 2 channels, a softmax function applied to the last layer yields the pixel wise predictions.

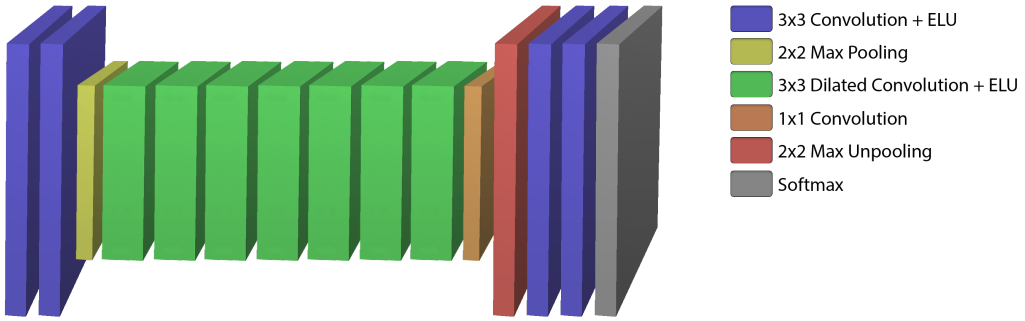


Figure 3.5: A visualization of the network architecture from [9], all standard convolutional layers have 32 channels and all dilated convolutions have 128 channels.

Layer	1	2	3	4	5	6	7
dilation factor (width, height)	(1,1)	(1,2)	(2,4)	(4,8)	(8,16)	(16,32)	(32,64)

Table 3.5: Dilation factors used in the dilated convolutions in the ContextNet architecture.

When trained in the GAN framework a modified version of this ContextNet was used as the encoder. The network was cut after the 1x1 convolutional layer, six stride 2 convolutional layers were added to downsample the features before mapping to latent space with a fully connected layer. The generator and discriminator architectures used were DCGAN style networks, the same networks used when training the SegNet architecture encoder. A latent space with 200 dimensions was used. In the supervised stage, the weights of the six additional convolutional layers were

discarded and the remaining weights were used to initialize the encoder and context module.

The architecture was also implemented in a VAE model. As an encoder the architecture up to the 1x1 convolution was included. Then six layers of convolution layers with stride 2 was added for down-sampling, giving a feature volume of size 13x7x512. Two parallel dense layers were then added for mapping to the latent space. The two dense layers gives the mean ($\boldsymbol{\mu}$) and the standard deviation ($\boldsymbol{\sigma}$) of the distribution for the latent variable. The dimension of this layer was set to be 100. The decoder was implemented using the same architecture as the DCGAN generator in the GAN model. This model was trained in a VAE setting until convergence. The trained encoder was then cut after the 1x1 convolution layer and used as the encoder in the original model of the context network in figure 3.5. This was then trained in a supervised mode for the semantic segmentation task.

3.6.4 Next frame prediction

Feature learning with next frame prediction was evaluated on the ContextNet architecture. When performing separate training, the training procedure was as follows: The entire network was trained on next frame prediction, the last two convolutional layers were then randomly reinitialized and trained to convergence on road segmentation, keeping earlier layers fixed. Finally, the entire model was fine tuned.

When performing joint training, two separate instances of the last two convolutional layers were maintained, one for next frame prediction and one for road segmentation. The network hence had one encoder and two decoder "heads". The two tasks were then trained in an alternating fashion, allowing both to propagate weight updates back through their respective decoders and all earlier (shared) layers.

3.7 Development & Implementation

The work of developing and implementing the methods in this thesis was done using the programming language Python™. Many of the methods used are computational demanding when implemented, especially training of the networks containing many parameters and large feature volumes such as DCGAN. The large number of parameters and feature volumes result in large memory requirements, and the forward and backward passes through the network are computationally expensive. To build an effective implementation of the methods studied in this thesis, the machine learning framework TensorFlow™ was used. The framework is developed mainly for training of neural networks and has support for using Graphics Processing Units (GPU), which made it suitable for this thesis. All the neural networks studied in this thesis were trained using a Nvidia Tesla P100 GPU provided by Autoliv.

4

Handwritten Digits Results

As proof of concept the generative models under considerations were tested on the MNIST handwritten digits dataset, a significantly simpler and lower resolution dataset than KITTI road. The MNIST dataset served as a problem of lower computational complexity, which allowed for quick experimentation and evaluation of ideas. Since the studied generative models are generally not applied to high resolution images, the MNIST tests also allowed for the isolation of which issues that arose from using higher resolution data. This chapter first covers various experiments to evaluate the different GAN formulations and VAE, and presents some qualitative results. Then, the results from using the studied generative models for unsupervised feature learning are presented.

4.1 Generative Models

The generative models were implemented and applied to the MNIST dataset. The models were evaluated by qualitative inspection of the generated samples.

4.1.1 GAN

The MNIST digits data were reshaped to a vector of 784 values, making the problem permutation invariant(no spatial information is used). To evaluate the different GAN formulations, a very simple GAN model was implemented with an architecture and hyperparameters inspired by [5]. The discriminator and generator architectures both consist of two 1000 neuron fully connected layers followed by an output layer. The generator uses leaky rectifier activations on the hidden layers and a tanh squashing function on the output layer. The discriminator uses standard rectifier activations on the hidden layers and a sigmoid on the output unit. Both D and G use batch normalization before the second activation function.

The model was trained using the standard GAN, WGAN and WGAN-GP formulations. In all cases the model was trained for 100 epochs and 5 discriminator training

steps were performed for each generator training step. A latent space with 50 dimensions and weight decay factor of $\lambda_{WD} = 0.00025$ were used. In the standard GAN formulation, the network was trained with ADAM optimization and a learning rate of 10^{-4}. When training using the WGAN formulation, the discriminator weights were clipped to [-0.01, 0.01] and RMSprop optimization with a learning rate of $5 \cdot 10^{-5}$ was used. The same network was also trained using the WGAN-GP formulation. The ADAM optimizer with learning rate 10^{-4} and momentum $\beta_1 = 0.5$ and $\beta_2 = 0.9$ was then used, and the gradient penalty scaling was set to $\lambda = 10$. In addition, the batch normalization in the discriminator was exchanged for layer normalization for this run, since discriminator batch normalization makes the Improved Wasserstein GAN formulation invalid. Some generated samples for all models can be seen in figure 4.1.1

Generator Inverse

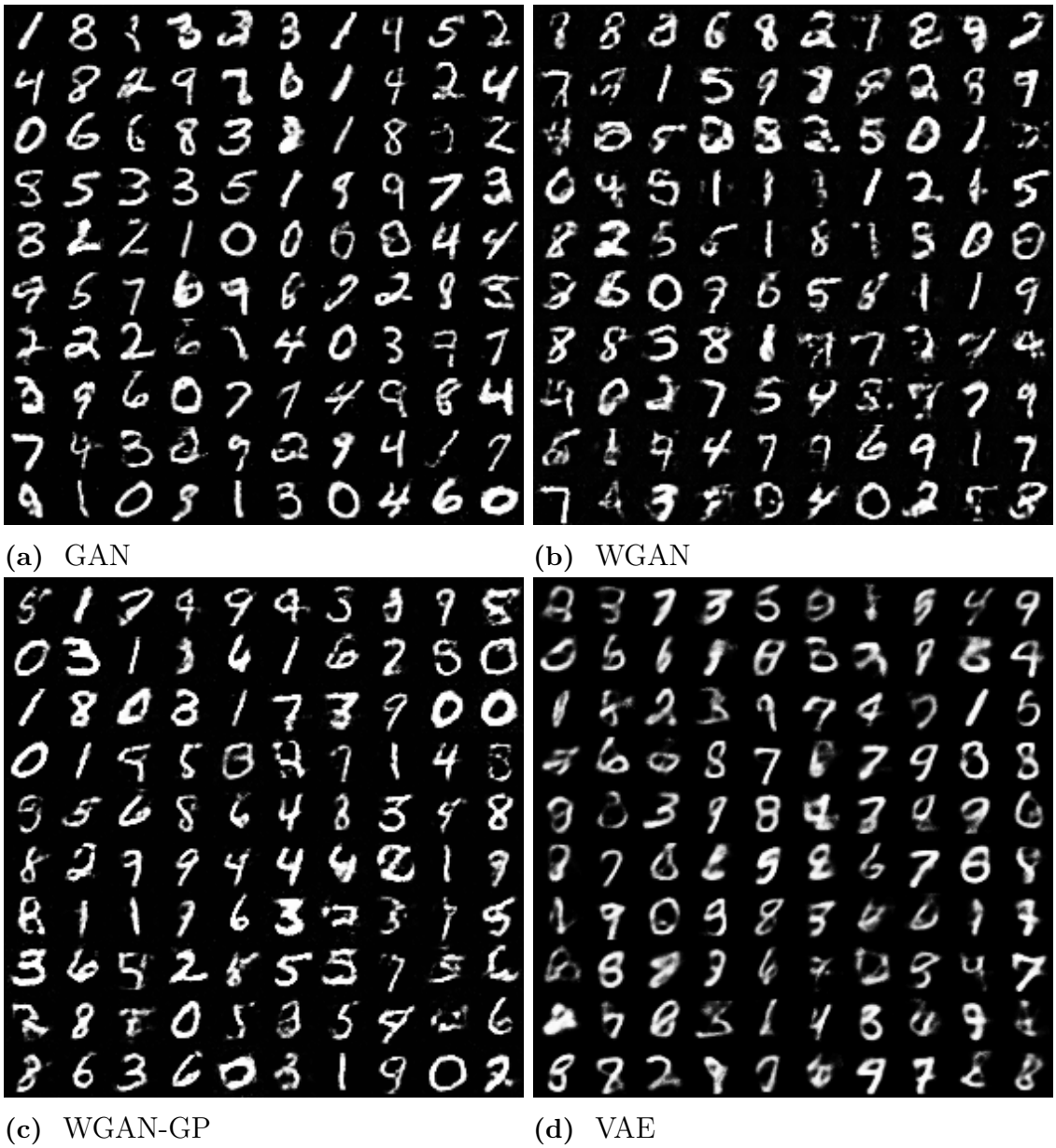
Two different methods for learning an inverse to the generator were evaluated, bidirectional GAN and latent regressor. The model was extended to a BiGAN by adding an encoder E, consisting of a fully connected neural network with 2 hidden layers with ReLU activations, batch norm before the second activation and a linear output layer.

Similarly to the previous GAN experiments, all models were trained for 100 epochs and with 5 discriminator updates for each generator update. The network was again trained using the GAN, WGAN and WGAN-GP formulations. The same hyper parameters and architecture modifications as for the GAN experiments were used. Some encoded and reconstructed MNIST digits for all models can be seen in figure 4.2.

Training the encoder as an latent regressor was also evaluated by simply exchanging the encoder loss function for the reconstruction loss in latent space. This was done in combination with the WGAN-GP model since this yielded the most promising results, some example output can be seen in figure 4.2.

4.1.2 VAE

The generative model using the VAE method was implemented with a similar architecture as the GAN model and [5]. The encoder was implemented using two fully connected layers with rectifier activation as hidden layers. As output of the encoder two parallel fully connected layers with linear activation was used to obtain the vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ for the distribution of the latent variable. As the decoder part of the VAE two fully connected layers with rectifier activation was used as hidden layers. The output layer was implemented by a fully connected layer with an output size of 784 and a sigmoid activation.

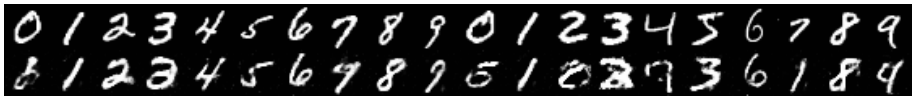


(a) GAN **(b)** WGAN

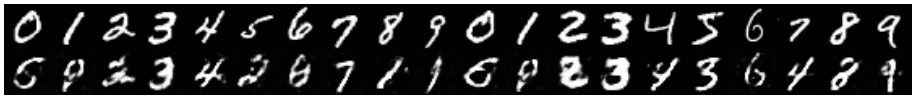
(c) WGAN-GP **(d)** VAE

Figure 4.1: Example of generative samples from the MNIST dataset. All models were trained for 100 epochs. The images were created by sampling from the latent variable and feeding the samples through the generator.

The model was trained using 1000 neurons in each hidden layer and a latent dimension of 20. The variational lower bound was used as the loss function and a ADAM optimizer with an initial learning rate of 10^{-4} was used. The model was trained for 100 epochs and qualitative results of the generated samples are shown in figure 4.1.1.



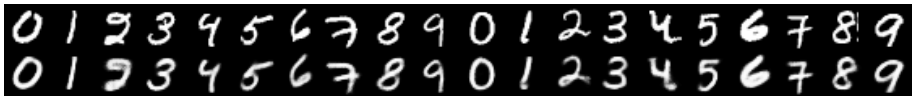
(a) bidirectional GAN



(b) bidirectional WGAN

(c) bidirectional WGAN-GP

(d) latent regressor WGAN-GP



(e) VAE

Figure 4.2: MNIST digits recreated by various BiGAN variants and VAE. The top row is X and the bottom row is the reconstructed image from $G(E(X))$

.

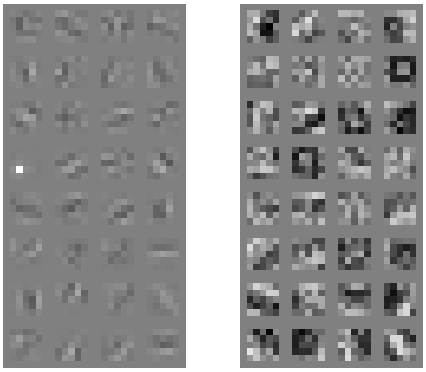
4.2 Unsupervised feature learning

The setups that yielded the most promising results during the generative model comparison were then also evaluated quantitatively by using the learned features on a semantic segmentation task. To do this the models were trained with an architecture more suitable for semantic segmentation.

4.2.1 Generative Stage

In the generative stage VAE and GAN models were trained on the MNIST data using the architectures described in section 3.5.1. In the GAN setting the WGAN-GP formulation with latent regressor was used. The model was trained for 400 epochs with ADAM optimization and a learning rate of 10^{-4}. The gradient penalty factor was set to $\lambda = 10$, and 5 discriminator updates were performed for each generator update. The filters learned in the first convolutional layer of the encoder using this procedure can be seen in figure 4.3.

The VAE model was trained with the architecture described in section 3.5.1. The model was trained for 100 epochs with a ADAM optimizer and learning rate of 10^{-4}. The encoder part of the trained model was then used as the encoder in the the supervised model.



(a) WGAN-GP latent regressor **(b)** VAE

Figure 4.3: Visualization of the filters in in the first CNN layer after training the encoder using unsupervised feature learning.

4.2.2 Supervised Stage

To evaluate the representations learned by the different approaches a simple experiment was constructed. An artificial semantic segmentation task was constructed from MNIST data following the procedure outlined in section 3.2.1.

The architecture used is outlined in figure 3.4, a dropout rate of 25% was used between all decoder layers. A fully supervised baseline was established by training the entire network from randomly initialized weights using only 500 labeled examples. This was then compared to a procedure when the encoder was initialized from weights learned in generative models, and only the decoder was trained supervised, as well as a procedure where the entire network was fine-tuned after the decoder had converged. The resulting test set performance for different models can be seen in table 4.1.

	Accuracy	IoU
Supervised	0.960	0.716
WGAN-GP latent regressor (fixed encoder)	0.956	0.743
WGAN-GP latent regressor (fine-tuned)	0.963	0.750
VAE (fixed encoder)	**0.989**	0.736
VAE (fine-tuned)	0.962	**0.768**

Table 4.1: Comparison of segmentation performance with and without unsupervised feature learning.

5

Road Segmentation Results

This chapter presents the results achieved on the KITTI road dataset. Both VAE and GAN based generative model were trained on the bird's-eye view lidar images described in section 3.1.2., as well as a model utilizing next-frame prediction as an auxiliary task for feature learning. The results from the generative models are presented first, followed by the results of the auxiliary task training. The section on generative models is divided into generative results and road segmentation results. The Generative results section presents qualitative results from evaluating the generative models and their learned feature representations, and the road segmentation results section covers quantitative and qualitative results of the performance impact of using the the generative feature representations when training models for road segmentation.

5.1 WGAN-GP with latent regressor

Out of the various GAN models and architectures evaluated the most promising results were achieved using WGAN-GP with latent regressor. The regular GAN and BiGAN models evaluated suffered from mode collapse, producing samples from only a small subset of the data distribution. The bidirectional WGAN-GP models converged and produced visually reasonable samples, but failed to produce reconstructions indicating that the encoder had captured important features of the data. Training the encoder as a latent regressor yielded better reconstructions. Hence, only the results from WGAN-GP with latent regressor are presented here.

5.1.1 Generative results

The architecture described in table 3.4 was trained on only the mean height channel of the lidar representation of the KITTI data (see sec 3.2.2). The model was trained for 50 epochs using ADAM optimization with a learning rate of 10^{-4}, gradient penalty scaling $\lambda = 10$ and 5 discriminator steps per generator step. Some example images generated by this model can be seen in figure 5.1.

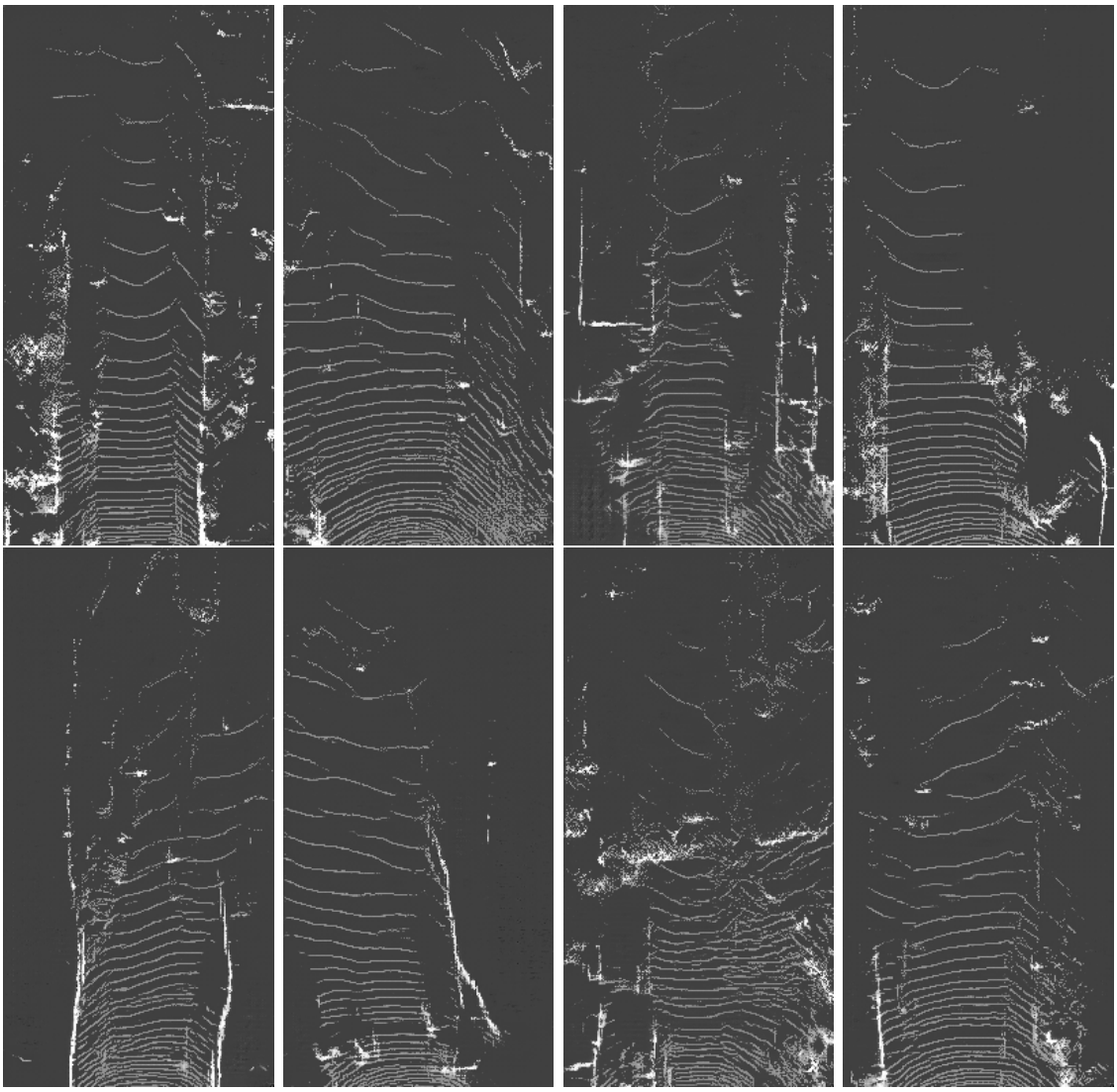
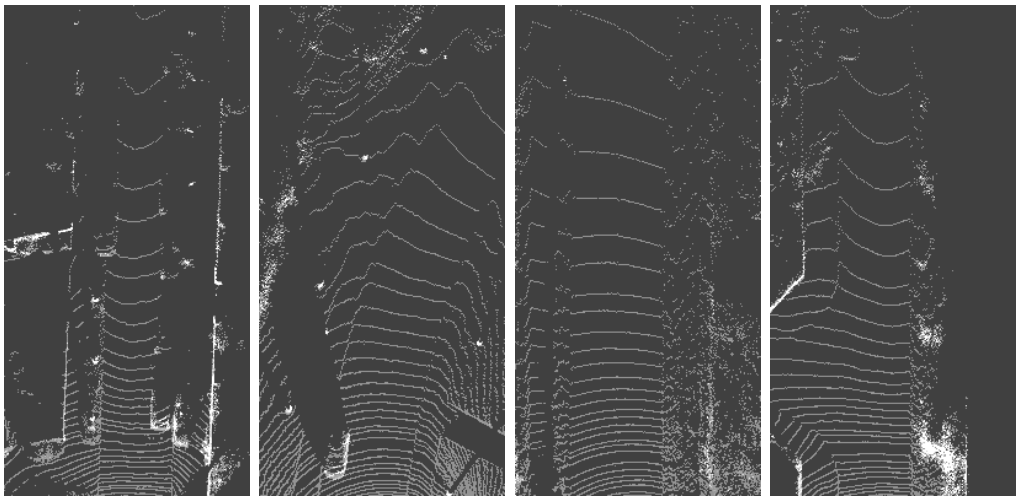


Figure 5.1: Example images generated from WGAN-GP trained with the modified DCGAN architecture on the KITTI dataset.

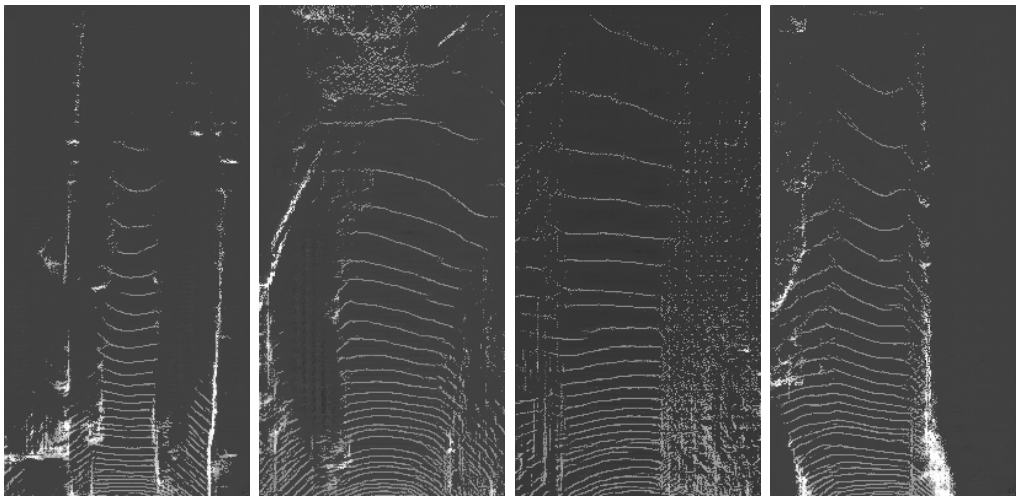
Because of the significantly larger models utilized, the latent regressor was not trained jointly with the GAN model but instead in a separate step once the generator had converged. This process requires less parameters and feature volumes loaded in memory at once and hence allows for training of larger models.

VGG latent regressor

A neural network following the VGG-16 architecture, see encoder in table 3.3, was trained as a latent regressor. The network was paired with the generator from the previous section. The latent regressor was trained for 50 epochs using ADAM optimization and a learning rate of 10^{-4}. Pairs of encoded and decoded lidar grids generated by this model can be seen in figure 5.4. While the reconstruction is not perfect it captures most important aspects of the images.



(a) encoded images $\boldsymbol{x}$



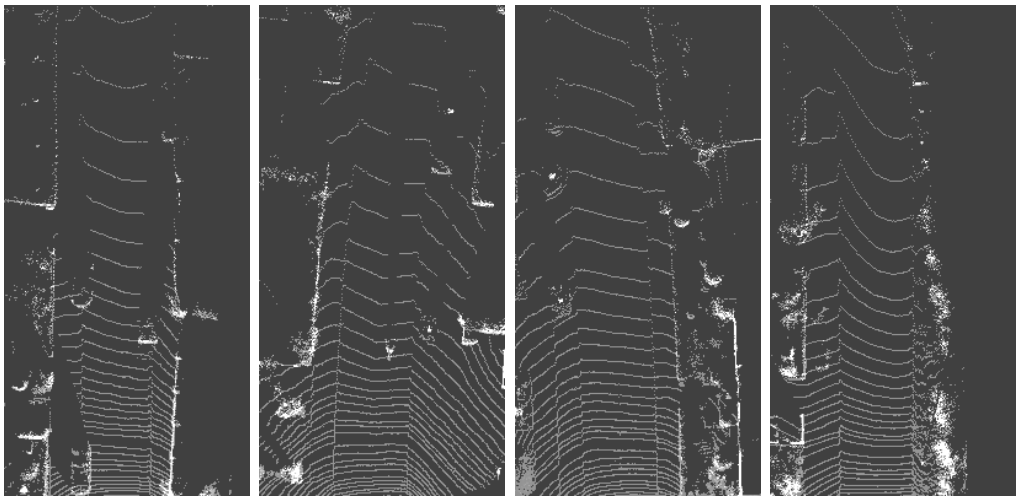
(b) decoded images $G(E(\boldsymbol{x}))$

Figure 5.2: Encoded-decoded image pairs from VGG encoder trained as an latent regressor with the modified DCGAN architecture trained using WGAN-GP.

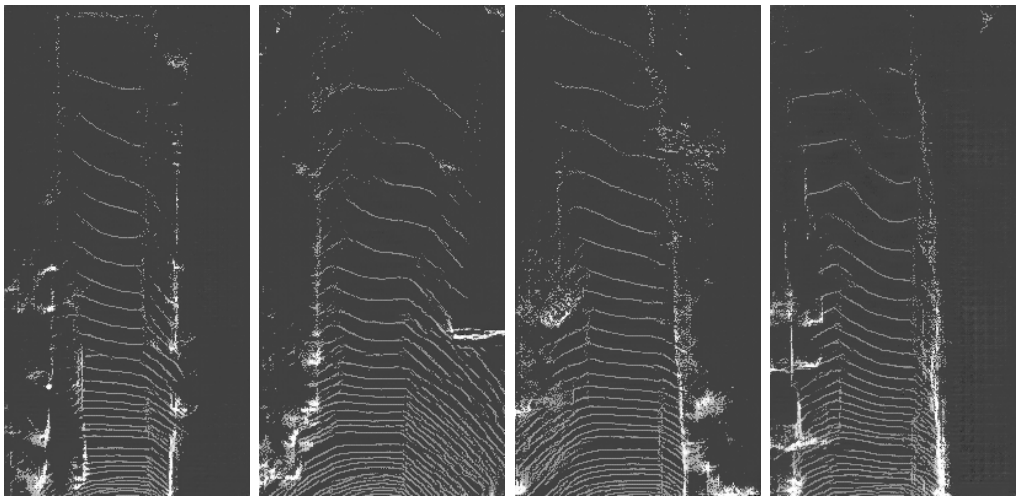
ContextNet latent regressor

The modified ContextNet encoder architecture covered in section 3.6.3 was trained as a latent regressor. This was done to allow for comparison to the ContextNet supervised baseline. The same generator as for the VGG based latent regressor was used.

The network was trained for 100 epochs using ADAM optimization and a learning rate of 10^{-4}. Some examples of encoded and reconstructed images can be seen in figure 5.3. The encoding captures important aspects of the image, but overall produces a slightly worse reconstruction than the VGG-based encoder. This is expected since the network is significantly smaller.



(a) encoded images $\boldsymbol{x}$



(b) decoded images $G(E(\boldsymbol{x}))$

Figure 5.3: Encoded-decoded image pairs from the ContextNet encoder trained as an latent regressor with WGAN-GP.

5.1.2 Road segmentation results

This section covers the comparisons of performance between randomly initialized networks and networks initialize from pretrained features, on the task of road segmentation. First the evaluations using the ContextNet architecture are presented, followed by the evaluations on the Segnet based architecture.

ContextNet network

The ContextNet network was trained with encoder and context module (layers 1 to 11) features learned from WGAN-GP with latent regressor. The encoder was fixed while the decoder was trained for 4 epochs, the entire network was then fine-tuned

	Fmax	Precision	Recall
Random initialization	0.935	0.933	0.937
Learned Features WGAN-GP (fine-tuned)	**0.943**	**0.951**	**0.942**

Table 5.1: Comparison of supervised and semi-supervised performance on road segmentation using the ContextNet architecture.

for 700 epochs. A learning rate of 10^{-3} was used. This was then compared to training the entire network from randomly initialized weights. The scores of the two methods on various evaluation metrics can be seen in table 5.1.

In addition to achieving slightly better performance, the network converges significantly faster when trained from learned features. Some examples of road segmentation provided by models trained from random initialization and learned features can be seen in figure 5.4, the samples are from the validation data.

SegNet Style network

The encoder of the SegNet network, described in section 3.6.1, was implemented as the encoder part of a generative network for the models WGAN-GP and VAE. These models were trained to generate credible samples. After convergence in the generative stage the trained encoder part of the network was used in a SegNet network performing semantic segmentation. The performance of these networks was then compared to a network were the encoder was instead initialized using weights pre-trained on ImageNet image classification [29], as well as randomly initialized weights.

The supervised network was trained using the annotated KITTI data. The encoder part of the network was fixed during the first 10 epochs of training where only the decoder was trained with a learning rate of 10^{-4}. After 10 epochs the learning rate was decreased to $5 \cdot 10^{-5}$ and the entire network of encoder and decoder was trained. To preventing overfitting a dropout layer with rate of 25% was applied after each pooling layer.

The results from applying this architecture to the road segmenation task were inconsistent and generally poor. No model achieved a f_{max}-score above 75%, which is substantially worse than the other models studied in this thesis. In addition, the results varied significantly between different experimental runs. The conclusion from the experiments is that the modified SegNet architecture could not be trained on the relatively small KITTI dataset in a satisfactory way. This also made the comparison between supervised and unsupervised feature learning inconclusive.

5.2 VAE

For the KITTI road detection task a VAE model was implemented using the ContextNet architecture described in section 3.6.3. The model was applied to the unannotated KITTI dataset explained in section 3.2.2. Different variations of the configuration of the input data were tested with different parameter settings for the model. The results showed that the model had difficulties to handle input features of continuous values, such as height features. Using these input the model quickly collapsed into a state of producing outputs with uniform feature values. When the model instead was applied to data using cell occupancy as the features it showed better performance, producing credible samples.

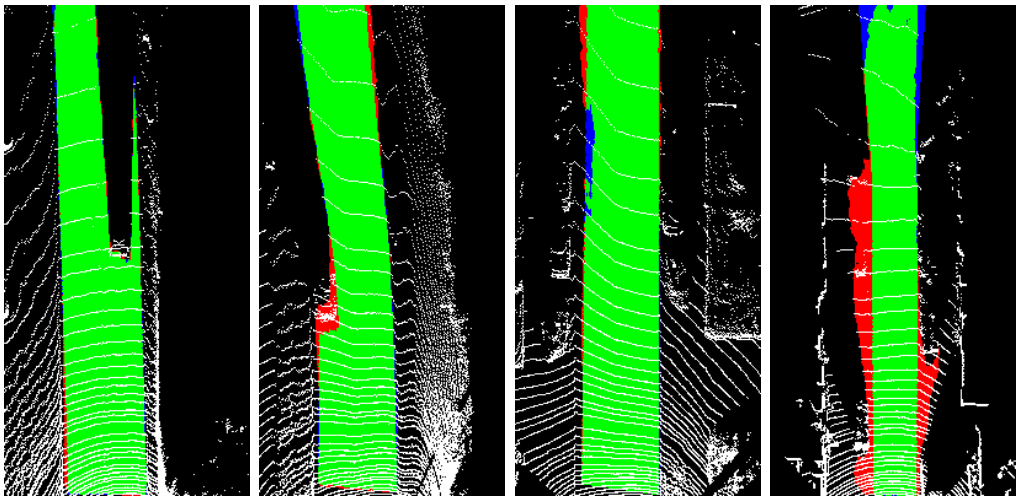
The generative model was trained for 20 epochs using the unannotated KITTI dataset. ADAM optimization with a learning rate of 10^{-4} was used to minimize the variational lower bound during the training. No dropout was used in the training. The unannotated dataset was considered to be large and diverse enough for the trained model to generalize well. In figure 5.5 a qualitative result is shown for the reconstruction of the images, and samples generated from the latent variable is shown in figure 5.6.

The encoder parameters learned in the generative stage were then used as initialization when training the ContextNet architecture on the road segmentation task. The input was replaced to use the mean height from each pixel as features and the samples was fed through the trained encoder of the generative model. From the last dilated convolution layer the output was retrieved, which was then fed through the decoder of the ContextNet to get an output of pixelwise class prediction. Training of the model was done using the annotated KITTI dataset. As loss function the cross-entropy was used, which was minimized using an ADAM optimizer with initial learning rate of 0.001. To prevent overfitting a dropout of 25% was used in the model.

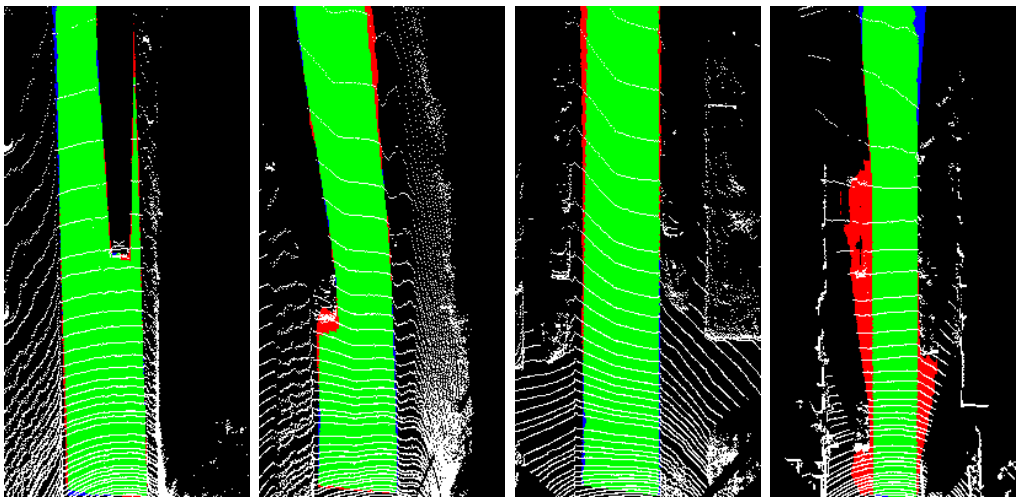
The model was evaluated by a comparison to a model where the encoder was initialized with random weights. When initializing the encoder from parameters learned in the VAE model a faster convergence for the segmentation task was observed. The performance of the model did show a marginal improvement compared to the model with randomly initialized weights. The performance of the different methods is shown in table 5.2 and some examples of segmentation can be seen in figure 5.4c.

	Fmax	Precision	Recall
Random initialization	0.935	0.938	0.936
Learned Features	**0.942**	**0.948**	**0.940**

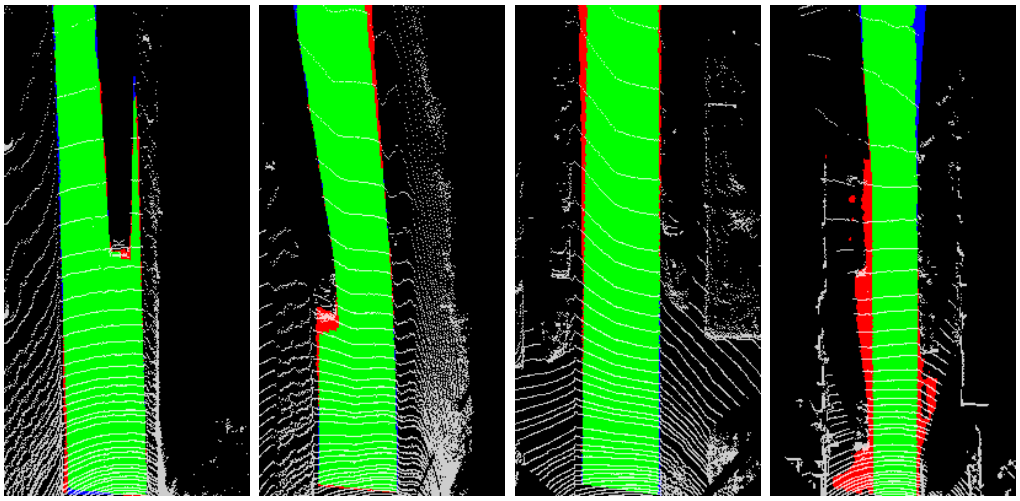
Table 5.2: Comparison of supervised and semi-supervised performance on road segmentation. The semi-supervised model is a VAE trained for 20 epochs on unannotated data.



(a) Fully supervised training

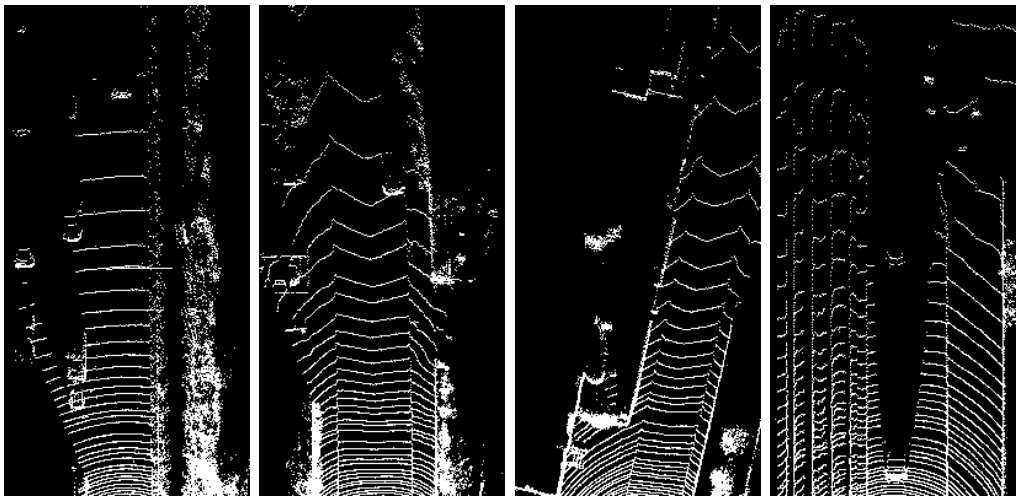


(b) Encoder initialized from latent regressor features

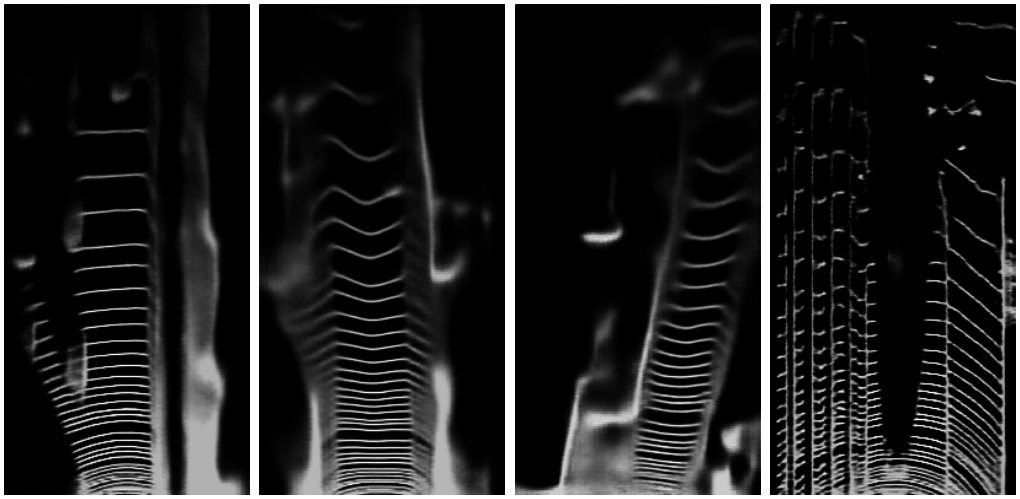


(c) Encoder initialized from trained VAE model.

Figure 5.4: Road segmentation examples with and without pre-trained features. Green represents true positive, blue represents false negative and red represents false positive pixel classifications.



(a) Original Images



(b) Reconstructed images

Figure 5.5: Reconstructed images of a VAE model trained for 20 epochs using ContextNet architecture. Top figures show the original images as input to the model. Bottom figures is the images reconstructed from the latent variable of the posterior distribution of each input image.

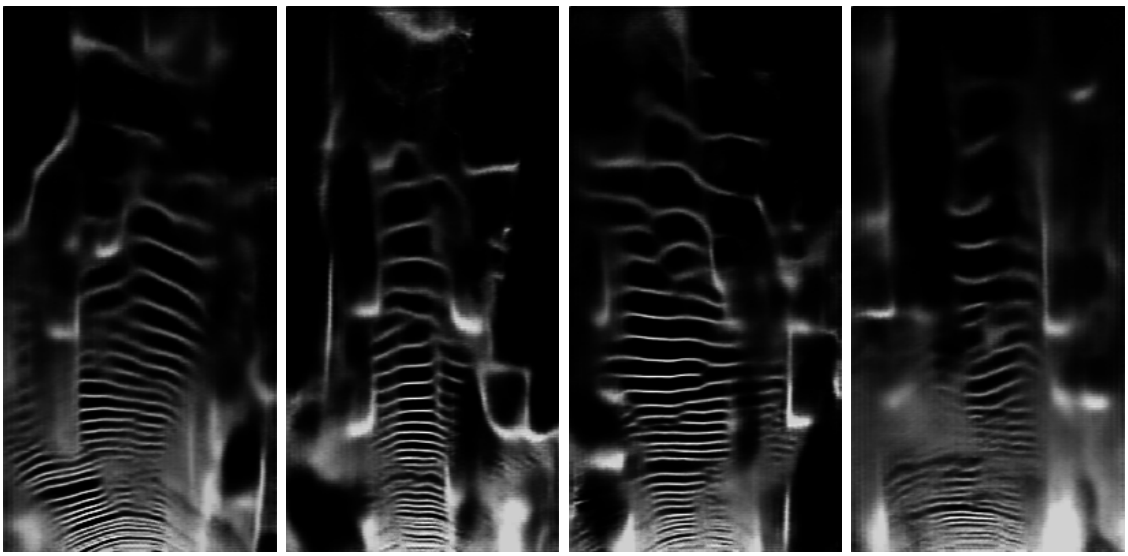


Figure 5.6: Generated samples from a VAE model trained for 20 epochs using the ContextNet architecture. A latent variable was sampled from a standard Gaussian distribution and fed through the decoder to generate an image.

5.3 Next frame prediction as auxiliary task

Feature learning by predicting the next frame was evaluated on the ContextNet architecture (figure 3.5). Since the network was trained using several frames as input, a separate supervised baseline was established by training the network from randomly initialized weights on only road segmentation. The network was again trained for 800 epochs with ADAM optimization and a learning rate of 10^{-3}.

For the separate training process, the network was first trained for 10 epochs on the next-frame prediction task, layers before the upsampling layer were then fixed and the remaining layers were trained to convergence on the road segmentation task. Finally, the entire network was fine tuned on the road segmentation task. The road segmentation performance after convergence can be seen in table 5.3.

For the joint training approach, the network was split at the unpooling layer, i.e. previous layers were shared while following layers were separate. The entire network was then trained alternating between the two tasks, the two loss components were evenly weighted, one next-frame prediction iterations were performed for every road segmentation iteration. The resulting performance can be seen in table 5.3.

	F_{max}	precision	recall
Fully Supervised (65 samples)	**0.923**	**0.938**	**0.920**
Next-frame, separate training (65 samples)	0.920	0.936	0.920
Next-frame, joint training (65 samples)	0.918	0.936	0.914
Fully Supervised (all samples)	**0.940**	**0.956**	**0.931**
Next-frame, separate training (all samples)	0.933	0.950	0.930
Next-frame, joint training (all samples)	0.933	0.953	0.925

Table 5.3: Comparison of performance on the KITTI road segmentation task with and without auxiliary task training.

6

Discussion

This chapter presents discussion and analysis of the results. First, the methods for evaluating the generative models are discussed. Then follows an analysis of the road segmentation performance, and what it tells us about the studied methods for unsupervised feature learning. In addition, a short note on our data processing is presented. Finally, the conclusions of this work are presented.

6.1 Generative Models

In this thesis, the main focus has been to investigate generative models. The models were evaluated by various methods, most of which focused on the generative capabilities of said models. This analysis was done using both quantitative metrics and qualitatively, with visual inspection. The reconstruction loss could be utilized to evaluated the models ability to reconstruct images. But for the models ability to generate credible samples from a distribution of latent variables, the analysis was more challenging. The samples could be visually inspected, and in this way it could be determined if they resembled the studied data. But this method did not enable us to evaluate that samples were generated from the entire distribution of data. There is hence a possibility that the trained models learned to generate samples only from a small subset of the data distribution. Another potential issue is that the model could overfit to the dataset and learn to simply recreate samples from the dataset. This could be have been measured by performing nearest neighbour analysis of a output sample and the data used, and assess how similar the samples are.

The intuitive idea behind using generative models in this thesis is to find feature representations from data. These features could then be transferred to be used in a model performing a sometimes totally different task, a task which is independent from the feature learning process of the generative model. Some tasks may benefit more from using the features than other tasks. How well the features learned from the structure of data itself transfer to different tasks is hard to evaluate quantitatively. But looking at the two problems of this thesis, it seems that the generative models could be better exploited in the handwritten digit segmentation task. In this problem the structure of the data itself is more similar to the targets of the

segmentation, compared to the road segmentation task where the structures are less similar. The choice of network architecture also likely has a significant influence on the impact of using pre-trained features. For instance, it is possible that if a well regularized network that naturally underfits the problem is used, the generalizing properties of pre-training might be neglected.

6.2 Results

The results show that unsupervised feature learning can marginally improve road segmentation performance using unlabeled data. This could prove to be a useful technique since unlabeled data exists in abundance. On the other hand, the performance improvements are on the order of $\sim 0.75\% f_{max}$. It is important to take into account that this is the performance gain on a dataset which consists of only 289 annotated images. It is possible that the performance gains from feature learning would be significantly less for the larger set of annotations that would be used in practice. It should be noted that the studied task is fairly simple and geometrical in its nature, and the supervised models achieve good performance and generalization even for the relatively small dataset used. It is possible that feature learning is more important for tasks were a semantic understanding of the data is important, such as when there are more classes to distinguish between. Generally, a problem involving more classes requires a larger dataset to train a model which generalizes well, meaning the regularizing properties of feature learning might have more impact in this setting.

In the generative models, it is important to note that it is not known whether or not the learned features are trained in an optimal fashion. There are a myriad of design choices in the generator and discriminator architectures and training procedures, which can only be quantitatively evaluated by using them in a supervised task. This results in a long feedback loop, and makes it hard to isolate variables. Hence these models could potentially have a larger impact with the correct training procedure. In the same way, the auxiliary task could have been stated in a variety of different ways. For instance, longer sequences could have been used, and the position of lidar detections could have been predicted instead of ground height. What drives a model to learn a good, general feature representation is not well known, and hence these models could potentially work significantly better.

6.3 Data processing

The data used in this thesis was obtained in a format which involved some preprocessing techniques [27]. The raw data from the Velodyne lidar sensor was not used directly but manually feature engineering was performed as described in section

3.2.2. Intuitively these features seem reasonable to use and are able to capture the states and appearances of the object in the receptive field of the sensor. But many of the features may be redundant and not necessary to include. This is indicated by the fact that using only some features representing the measurements shows similar performance as using all features. An interesting approach would be to remove the pre-processing and feature engineering from the data and instead use the raw lidar measurements as an input to a model. But the varying size of the measurements form different time steps causes a problem. Such a model would need to include recurrent machine learning methods. This is not included in the thesis and is left for future research.

6.4 Conclusion

In conclusion, our findings suggest that unsupervised feature learning through generative modelling could be useful for semantic segmentation tasks. The performance gains are marginal on the KITTI road segmentation task, but it is possible that the methods could work better with different design choices, or are more useful on tasks involving more classes.

Recent advances in training of neural networks, most notably batch normalization, spatial dropout and better architecture designs, have allowed deeper architectures to be trained from scratch. This means that, for many tasks, the utilization of unsupervised pre-training in most practical applications currently might not be an efficient solution, except for cases where annotated data is extremely prohibitive to create.

Being able to leverage unannotated data effectively to improve the capabilities of machine learning systems is a long standing research question. Recently, generative models have given raise to big advances towards being able to do this for classification problems, and some work has been done to extend this to semantic segmentation [5], [30]. Much is yet to be understood about what makes a feature representation work well for a certain task. Answering these questions will undoubtedly continue to be an exciting line of research in the future.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[2] Olga Russakovsky et al. "Imagenet large scale visual recognition challenge". In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

[3] Tim Salimans et al. "Improved Techniques for Training Gans". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2226–2234.

[4] Lars Maaløe et al. "Auxiliary Deep Generative Models". In: *arXiv preprint arXiv:1602.05473* (2016).

[5] Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. "Adversarial Feature Learning". In: *arXiv preprint arXiv:1605.09782* (2016).

[6] Luca Caltagirone et al. "LIDAR-based Driving Path Generation Using Fully Convolutional Neural Networks". In: *arXiv:1703.08987v2* (2017).

[7] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[8] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[9] Luca Caltagirone et al. "Fast LIDAR-based Road Detection Using Convolutional Neural Networks". In: *arXiv preprint arXiv:1703.03613* (2017).

[10] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. "Learning Deconvolution Network for Semantic Segmentation". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 1520–1528.

[11] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[12] Diederik P Kingma and Max Welling. "Auto-encoding Variational Bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[13] Gareth James et al. *An introduction to statistical learning*. Vol. 6. Springer, 2013. Chap. 2.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. Chap. 5, 8.

[15] Xavier Glorot and Yoshua Bengio. “Understanding the Difficulty of Training Deep Feedforward Neural Networks.” In: *Aistats*. Vol. 9. 2010, pp. 249–256.

[16] Fisher Yu and Vladlen Koltun. “Multi-scale context aggregation by dilated convolutions”. In: *arXiv preprint arXiv:1511.07122* (2015).

[17] Rajat Raina et al. “Self-taught Learning: Transfer Learning from Unlabeled Data”. In: *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 759–766.

[18] Ian J. Goodfellow. “NIPS 2016 Tutorial: Generative Adversarial Networks”. In: *CoRR* abs/1701.00160 (2017). URL: `http://arxiv.org/abs/1701.00160`.

[19] Carl Doersch. “Tutorial on Variational Autoencoders”. In: *arXiv preprint arXiv:1606.05908* (2016).

[20] Soumith Chintala Martin Arjovsky and L´eon Bottou. “Wasserstein GAN”. In: *arXiv:1605.09782v6* (2016).

[21] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised Representation lLearning with Deep Convolutional Generative Adversarial Networks”. In: *arXiv preprint arXiv:1511.06434* (2015).

[22] Luke Metz et al. “Unrolled Generative Adversarial Networks”. In: *arXiv preprint arXiv:1611.02163* (2016).

[23] Tong Che et al. “Mode Regularized Generative Adversarial Networks”. In: *arXiv preprint arXiv:1612.02136* (2016).

[24] Ishaan Gulrajani et al. “Improved Training of Wasserstein GANs”. In: *arXiv preprint arXiv:1704.00028* (2017).

[25] Martin Arjovsky and Léon Bottou. “Towards Principled Methods For Training Generative adversarial networks”. In: *NIPS 2016 Workshop on Adversarial Training. In review for ICLR*. Vol. 2016. 2017.

[26] Deepak Pathak et al. “Context encoders: Feature Learning by Inpainting”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2536–2544.

[27] Jannik Fritsch, Tobias Kuhnl, and Andreas Geiger. “A New Performance Measure and Evaluation Benchmark for Road Detection Algorithms”. In: *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on*. IEEE. 2013, pp. 1693–1700.

[28] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. “Segnet: A deep Convolutional Encoder-Decoder Architecture for Image Segmentation”. In: *arXiv preprint arXiv:1511.00561* (2015).

[29] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *CoRR* abs/1409.1556 (2014).

[30] Nasim Souly, Concetto Spampinato, and Mubarak Shah. “Semi and Weakly Supervised Semantic Segmentation Using Generative Adversarial Network”. In: *arXiv preprint arXiv:1703.09695* (2017).

A

Appendix

A.1 Kullback-Leibler for Gaussian Distributions

When computing the variational lower bound to be maximized in the VAE case, a term of Lullback-Liebler divergence is included. This term can in some cases be computed analytically depending on the distributions. For this case the distributions are

$$\begin{aligned} q_\phi(\mathbf{z}|\mathbf{x}) &= \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_\phi(\mathbf{x}), \boldsymbol{\sigma}_\phi(\mathbf{x})) \\ p(\mathbf{z}) &= \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}) \end{aligned} \tag{A.1}$$

and the divergence is given by

$$D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) = \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} d\mathbf{z} = \int q_\phi(\mathbf{z}|\mathbf{x})(\log q_\phi(\mathbf{z}|\mathbf{x}) - \log p(\mathbf{z}))d\mathbf{z}. \tag{A.2}$$

Computing the integrals gives

$$\begin{aligned} \int q_\phi(\mathbf{z}|\mathbf{x}) \log q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} &= -\frac{d}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{d}(1 + \log \sigma_j^2) \\ \int q_\phi(\mathbf{z}|\mathbf{x}) \log p(\mathbf{z}) d\mathbf{z} &= -\frac{d}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{d}(\mu_j^2 + \sigma_j^2) \end{aligned} \tag{A.3}$$

where d are the dimension of the distributions, μ_j the j:th element of the mean vector and σ_j the j:th elements of the variance vector. Adding the equations the Kullback-Liebler divergence is given by

$$D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) = \frac{1}{2}\sum_{j=1}^{d}(\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1). \tag{A.4}$$

From this expression the gradients of the parameters $\boldsymbol{\phi}$ contained in the μ_i and σ_i can be obtained.